Taruvi Cloud Introduction

Taruvi Cloud is a comprehensive Backend-as-a-Service (BaaS) platform designed specifically for AI agent development. Similar to Supabase, it provides a complete backend infrastructure but with a key differentiator: **enforced access control based on intelligent data models**.

Platform Architecture

Multi-Tenant Structure with SaaS Features

- **Sites**: Users create sites on taruvi.cloud, each generating a dedicated tenant using Django-tenants architecture
- **Apps**: Within each site, users can create and manage multiple applications
- **Configuration**: Each app supports configurable schemas, functions, proxies, and API tokens

Site-Level Features

Core Infrastructure

- **Authentication systems** with customizable providers (OAuth, SAML, JWT)
- **Domain management** for custom branding and routing
- **Frontend workers** for distributed processing and edge deployment
- **SSL/TLS management** with automatic certificate provisioning

SaaS & Billing Features

- **Subscription Management**: Tiered pricing plans (Starter, Pro, Enterprise)
- **Usage Tracking**: Real-time monitoring of API calls, storage, compute resources
- **Billing Integration**: Automated invoicing with Stripe/PayPal integration
- **Resource Quotas**: Configurable limits on databases, functions, API calls
- **Analytics Dashboard**: Usage metrics, performance insights, cost breakdowns
- **Team Management**: Multi-user access with role-based permissions
- **Audit Logging**: Comprehensive activity tracking for compliance

App-Level Features

Data Management

- **Dynamic Schemas**: JSON Schema-based table definitions with validation
- **Real-time Database**: PostgreSQL with real-time subscriptions
- **File Storage**: S3-compatible object storage with CDN integration

- **Search Engine**: Full-text search with Elasticsearch integration
- **Data Migrations**: Version-controlled schema changes with rollback support

Function Runtime

- **Serverless Functions**: Python runtime with package management
- **Edge Functions**: Distributed execution at edge locations
- **Scheduled Jobs**: Cron-like scheduling with retry mechanisms
- **Event Triggers**: Database, HTTP, and time-based triggers
- **Integration Proxies**: No-code connectors to n8n, Zapier, Make

Security & Access Control

- **RBAC System**: Fine-grained role and permission management
- **Policy Engine**: Row-level security with dynamic filtering
- **Data Masking**: Automatic PII detection and masking
- **Rate Limiting**: Configurable throttling per endpoint/user
- **API Key Management**: Scoped tokens with expiration

API & Integration

- **REST API**: Auto-generated CRUD endpoints
- **GraphQL**: Dynamic schema generation with subscriptions
- **WebSocket**: Real-time connections with authentication
- **Webhook Management**: Outbound event notifications
- **SDK Generation**: Auto-generated client libraries

MCP Server Integration & AI Agent Workflow

MCP Server Architecture

Each app automatically provisions a dedicated **Model Context Protocol (MCP) server** that acts as the intelligent interface between AI agents and the Taruvi Cloud backend.

AI Agent Interaction Flow

1. Schema Management

```
# AI Agent creates/modifies schemas via MCP
mcp_client.create_table({
    "name": "users",
    "fields": {
        "id": {"type": "uuid", "primary": true},
        "email": {"type": "string", "unique": true, "pii": true},
        "profile": {"type": "jsonb", "searchable": true}
    },
    "policies": ["authenticated_read", "owner_write"]
})
```

2. Function Deployment

```
# AI Agent deploys serverless functions
mcp_client.create_function({
    "name": "send_welcome_email",
    "trigger": "database.users.insert",
    "code": """
    def handler(event, context):
        user = event['record']
        send_email(user['email'], template='welcome')
        return {'status': 'sent'}
    """,
    "permissions": ["email:send"]
})
```

3. Access Control Creation

```
# AI Agent sets up RBAC
mcp_client.create_role({
    "name": "content_editor",
    "permissions": [
        "posts:read",
        "posts:create",
        "posts:update:own"
    ],
    "data_filters": {
        "posts": "author_id = current_user.id"
    }
})
```

4. Policy Implementation

```
# AI Agent creates data policies
mcp_client.create_policy({
    "name": "gdpr_compliance",
    "tables": ["users", "user_activity"],
    "rules": {
        "pii_masking": true,
        "retention_days": 730,
        "export_format": "json",
        "deletion_cascade": ["user_sessions", "user_preferences"]
    }
})
```

5. Integration Setup

```
# AI Agent configures external integrations
mcp_client.create_proxy({
    "name": "crm_sync",
    "type": "n8n",
    "workflow_id": "user_registration_flow",
    "triggers": ["users.insert", "users.update"],
    "auth": {"type": "api_key", "key": "{{secrets.n8n_api_key}}"}
})
```

MCP Server Capabilities

Real-time Schema Introspection

- AI agents can query current schema state

- Automatic dependency resolution for schema changes
- Impact analysis before modifications

Intelligent Code Generation

- Auto-generates API endpoints from schema
- Creates validation rules based on data types
- Generates client SDK code in multiple languages

Security Policy Enforcement

- Real-time policy validation during schema changes
- Automatic security audit recommendations
- Compliance checking (GDPR, HIPAA, SOC2)

Resource Management

- Monitors usage against billing limits
- Auto-scaling recommendations
- Performance optimization suggestions

AI Agent Development Cycle

1. **Discovery**: AI agent introspects existing app structure via MCP
2. **Planning**: Agent analyzes requirements and proposes architecture
3. **Implementation**: Agent creates schemas, functions, and policies incrementally
4. **Testing**: Agent validates functionality through generated test endpoints
5. **Deployment**: Agent promotes changes through staging to production
6. **Monitoring**: Agent tracks performance and suggests optimizations

This architecture enables AI agents to not just consume backend services, but actively participate as full-stack developers, managing everything from database design to security implementation while maintaining enterprise-grade reliability and compliance.

Screenshots from initial demo at

`https://taruvi-5na.pages.dev/`

If accessing - signup first and then click on confirmation email once you receive one , only then go back and sign in

Site to site migration not allowed from frontend

App Promotion should be allowed - Transfer functionality Source → Target
Multiple users for a single organization needs to be there (django-organization)
Assign sites to users

Onboarding flow

User creates organization
User invites other users with the organization and restricts site access for user

Export app needed .
Starter Templates
Enable site by default for frontend platform
User management Day 0 at site level and cloud
Groups / RBAC Day 0

Environment variables in site management
Environment variables in App / Functions
Schema Versioning to be considered , Functions too django-history , django-reverse
Webhooks site level?
Place for templates - phase 2
Storage Module for Static Assets and others
Email templates and others also on storage module
PDF Mapping module - Check on html mapping flow
PDF Mapping should be inside functions as system functions
Authentication switcher on functions

Crontab for function execution schedule / triggers
Phase 2 planning : Events to be trigger functions based on app schema
Data exports on schema : to prioritize
Claude code document with context7 and other mcp servers
Marketplace for taruvi

Physical Architecture

Context

Taruvi Cloud requires a scalable, highly available, and cost-effective physical architecture that can support a multi-tenant Backend-as-a-Service platform with AI agent development capabilities. The platform must handle variable workloads, provide strong security isolation, support real-time features, and offer global distribution capabilities.
/

Requirements

- **Multi-Tenant Isolation**: Strong tenant isolation with shared infrastructure efficiency
- **High Availability**: 99.9% uptime SLA with multi-AZ deployment
- **Auto-Scaling**: Dynamic scaling based on demand with cost optimization
- **Global Distribution**: Low-latency access for worldwide users
- **Security**: Enterprise-grade security with compliance capabilities
- **Real-Time Features**: WebSocket support and real-time data synchronization
- **Background Processing**: Reliable job processing and queue management
- **Edge Computing**: Frontend deployment at edge locations
- **Monitoring**: Comprehensive observability and alerting
- **Cost Efficiency**: Optimize for variable workloads and startup economics

Current Constraints

- Django-based backend application
- PostgreSQL database requirement for multi-tenancy
- Need for background job processing (Celery)
- Real-time subscriptions and WebSocket support
- File storage and CDN requirements
- Compliance requirements (GDPR, SOC 2, etc.)

Decision

We will implement a **Cloud-Native Serverless Architecture** using AWS managed services with Cloudflare Workers for edge computing, optimized for multi-tenant SaaS workloads.

High-Level Architecture Overview

```
css
Copy code
Internet Users
    ↓
[Cloudflare CDN/WAF] → [AWS Route 53]
    ↓
[Application Load Balancer]
    ↓
[ECS Fargate Services]
    ↓
[Aurora PostgreSQL] + [ElastiCache Redis] + [SQS/Celery]
    ↓
[S3 Storage] + [CloudWatch Monitoring]
```

Infrastructure Components

1. Compute Layer - Amazon ECS with AWS Fargate

ECS Cluster Architecture

Service Separation:

- **Web Service**: Django application serving REST and GraphQL APIs
- **WebSocket Service**: Dedicated service for real-time connections and subscriptions
- **Worker Service**: Celery workers for background processing and scheduled tasks
- **MCP Service**: Model Context Protocol servers for AI agent interactions

Fargate Resource Allocation:

- **Web Service**: 512 CPU to 4096 CPU units with 1GB to 8GB RAM, auto-scaling 2-50 instances
- **WebSocket Service**: 256 to 1024 CPU units with 512MB to 2GB RAM, auto-scaling 2-10 instances
- **Worker Service**: 256 to 2048 CPU units with 512MB to 4GB RAM, auto-scaling 1-20 instances
- **MCP Service**: 256 to 1024 CPU units with 512MB to 2GB RAM, auto-scaling 1-10 instances

Container Strategy:

- **Base Images**: Python 3.11 slim images optimized for production
- **Multi-Stage Builds**: Separate build and runtime stages for smaller production images
- **Registry**: Amazon ECR with automated vulnerability scanning and lifecycle policies
- **Image Versioning**: Semantic versioning with automated tagging

Auto-Scaling Configuration

Scaling Triggers:

- **CPU-Based**: Scale out at 70% CPU utilization, scale in at 30%
- **Memory-Based**: Scale out at 80% memory utilization, scale in at 40%
- **Request-Based**: Scale based on requests per target threshold
- **Custom Metrics**: Database connection pool usage, queue depth, response latency

Scaling Behavior:

Scale-Out Cooldown: 5 minutes to prevent rapid scaling

- **Scale-In Cooldown**: 10 minutes for stability
- **Minimum Capacity**: Always maintain 2 instances for high availability
- **Maximum Capacity**: Cap at 50 instances with budget alerts

2. Database Layer - Amazon Aurora PostgreSQL

Aurora Cluster Configuration

Cluster Topology:

- **Writer Instance**: Single writer in primary AZ with automatic failover
- **Reader Instances**: 2-5 read replicas distributed across AZs with auto-scaling
- **Instance Classes**: db.r6g.large baseline scaling to db.r6g.2xlarge under load
- **Storage**: Aurora Serverless v2 with automatic storage scaling

Performance Optimization:

- **Aurora Capacity Units**: 0.5 minimum to 16 maximum ACUs
- **Connection Management**: RDS Proxy for connection pooling and failover
- **Read/Write Splitting**: Automatic routing of read queries to replicas
- **Query Performance**: Performance Insights for query optimization

Backup and Recovery:

- **Automated Backups**: 7-day retention with point-in-time recovery
- **Manual Snapshots**: Weekly snapshots with 30-day retention
- **Backup Windows**: 3:00-4:00 AM UTC during low traffic
- **Cross-Region Backups**: Automated replication to secondary region

Multi-Tenant Database Strategy

- **Schema-Per-Tenant**: Each site gets dedicated PostgreSQL schema
- **Connection Pooling**: PgBouncer integration for efficient connection management
- **Query Isolation**: Row-level security policies as backup isolation
- **Performance Monitoring**: Per-tenant query performance tracking

3. Networking and Load Balancing

Virtual Private Cloud (VPC) Design

Network Segmentation:

- **Public Subnets**: ALB and NAT Gateways across 3 AZs (10.0.1.0/24, 10.0.2.0/24, 10.0.3.0/24)
- **Private Subnets**: ECS services and application logic (10.0.10.0/24, 10.0.20.0/24, 10.0.30.0/24)
- **Database Subnets**: Aurora and ElastiCache isolated subnets (10.0.100.0/24, 10.0.200.0/24, 10.0.300.0/24)
- **CIDR Block**: 10.0.0.0/16 with room for expansion

Network Security:

Internet Gateway: Public subnet internet access

- **NAT Gateways**: Private subnet outbound internet access with high availability
- **VPC Endpoints**: Direct access to S3 and other AWS services without internet routing
- **Flow Logs**: VPC traffic monitoring for security analysis

Application Load Balancer (ALB)
Load Balancer Configuration:

- **Type**: Internet-facing ALB deployed across public subnets
- **Listeners**: HTTP (port 80) with automatic HTTPS (port 443) redirect
- **SSL Termination**: AWS Certificate Manager integration with automatic renewal
- **Health Checks**: Application-specific health endpoints with custom success criteria

Traffic Routing:

- **Web API**: Routes to ECS web service target group with sticky sessions disabled
- **WebSocket**: Routes to WebSocket service with sticky sessions enabled for connection persistence
- **MCP Service**: Dedicated path-based routing for AI agent interactions
- **Static Assets**: Direct routing to CloudFront distribution

DNS and Global Distribution
Route 53 Configuration:

- **Primary Domain**: taruvi.cloud hosted zone with DNSSEC
- **Custom Domains**: Customer domain support with validation workflows
- **Health Checks**: Multi-region endpoint monitoring with automatic failover
- **Geolocation Routing**: Route users to nearest available region

CloudFront CDN:

- **Origin Configuration**: ALB as primary origin with S3 as backup
- **Cache Behaviors**: API responses cached based on headers, static assets cached aggressively
- **Edge Locations**: Global distribution with 200+ locations
- **Security**: Origin Access Control for S3, custom headers for ALB access

4. Caching and Session Management

ElastiCache Redis Configuration
Cluster Architecture:

- **Engine**: Redis 7.0 with cluster mode enabled
- **Node Configuration**: cache.r6g.large instances with 3 primary nodes
- **Replication**: 1 replica per primary node for high availability
- **Sharding**: Automatic sharding across 3 shards for horizontal scaling

Security and Access:

- **Encryption**: TLS encryption in transit and at-rest encryption

- **Authentication**: Redis AUTH with token-based authentication
- **Network Isolation**: Deployed in private subnets with security group restrictions
- **Backup**: Daily automated backups with 7-day retention

Caching Strategy:

- **Session Storage**: User authentication tokens and session data
- **Application Cache**: Database query results and computed values
- **Rate Limiting**: API rate limiting counters and throttling data
- **Real-Time State**: WebSocket connection state and real-time subscription management

5. Message Queue and Background Processing

Amazon SQS Configuration

Queue Architecture:

- **Default Queue**: General background tasks with 5-minute visibility timeout
- **High Priority**: Critical tasks with 1-minute visibility timeout
- **Scheduled Jobs**: Time-based tasks with configurable delay
- **Dead Letter Queue**: Failed message handling with 14-day retention

Queue Management:

- **Message Retention**: 14 days for all queues with configurable per-queue settings
- **Batch Processing**: Support for batch message handling for efficiency
- **FIFO Queues**: Order-sensitive tasks with exactly-once processing
- **Cross-Region Replication**: Queue replication for disaster recovery

Celery Integration Architecture

Worker Types:

- **General Workers**: Default task processing with 4 concurrent processes
- **Heavy Workers**: CPU/memory intensive tasks with 2 concurrent processes
- **Scheduled Workers**: Cron-like tasks with single process per schedule
- **Priority Workers**: High-priority tasks with dedicated resource allocation

Task Management:

- **Result Backend**: Redis cluster for task result storage
- **Task Routing**: Intelligent routing based on task type and priority
- **Retry Logic**: Exponential backoff with configurable retry limits
- **Monitoring**: Task execution metrics and failure tracking

6. Storage and Content Delivery

Amazon S3 Storage Strategy

Bucket Organization:

- **User Uploads**: Per-tenant prefixes with versioning enabled
- **Static Assets**: Application assets with CloudFront integration
- **Backup Storage**: Database backups and application state snapshots
- **Log Archive**: Long-term log storage with lifecycle policies

Storage Optimization:

- **Intelligent Tiering**: Automatic optimization between storage classes
- **Lifecycle Policies**: 30-day Standard-IA transition, 90-day Glacier transition
- **Cross-Region Replication**: Critical data replication for disaster recovery
- **Compression**: Automatic compression for text-based content

Content Delivery Network

CloudFront Configuration:

- **Global Distribution**: Edge locations across 6 continents
- **Origin Shield**: Additional caching layer for origin protection
- **Custom Behaviors**: API-specific caching rules with TTL configuration
- **Real-Time Logs**: Request-level logging for analytics and debugging

Cache Strategy:

- **Static Assets**: 1-year cache with versioned URLs
- **API Responses**: 5-minute cache for GET requests with appropriate headers
- **Dynamic Content**: No cache for user-specific content
- **Purging**: Automated cache invalidation on content updates

7. Frontend Edge Computing

Cloudflare Workers (Primary)

Global Edge Network:

- **Locations**: 200+ edge locations worldwide
- **KV Storage**: Site configuration and routing data
- **Durable Objects**: Real-time collaboration and stateful edge computing
- **R2 Storage**: Static asset storage with global distribution

Worker Functionality:

- **Site Routing**: Dynamic routing based on custom domains
- **App Frontend**: Serve application frontends from edge locations
- **API Proxying**: Intelligent API request routing and caching
- **Authentication**: Edge-based authentication and authorization

Performance Benefits:

- **Cold Start**: Sub-10ms cold start times

- **Global Latency**: <50ms response times worldwide
- **Bandwidth**: Unlimited bandwidth included in pricing
- **Availability**: 99.99% uptime SLA

AWS Lambda@Edge (Alternative)
Edge Computing Alternative:

- **Runtime**: Node.js 18.x with 30-second timeout
- **Memory**: 128MB standard allocation
- **Triggers**: Origin request/response and viewer request/response
- **Integration**: Native CloudFront integration

Use Cases:

- **A/B Testing**: Edge-based traffic splitting
- **Personalization**: User-specific content modification
- **Security**: Request validation and filtering
- **Redirects**: Domain and URL redirection logic

8. Security Architecture

Network Security
AWS Web Application Firewall (WAF):

- **Managed Rules**: Common attack patterns, SQL injection, XSS protection
- **Rate Limiting**: 2000 requests per 5-minute window per IP
- **Geo-Blocking**: Country-based access restrictions if needed
- **Custom Rules**: Application-specific security rules

VPC Security Groups:

- **ALB Security Group**: Allow HTTP/HTTPS from internet (0.0.0.0/0)
- **ECS Security Group**: Allow traffic only from ALB security group
- **Database Security Group**: Allow PostgreSQL only from ECS security group
- **Redis Security Group**: Allow Redis only from ECS security group

Network Access Control Lists (NACLs):

- **Public Subnet NACLs**: Allow HTTP/HTTPS and ephemeral port responses
- **Private Subnet NACLs**: Allow internal communication and database access
- **Database Subnet NACLs**: Restrict to database traffic only

Identity and Access Management
IAM Role Strategy:

- **ECS Task Roles**: Minimal permissions for S3, SQS, and Secrets Manager access
- **ECS Execution Roles**: Container registry and log publishing permissions

- **Lambda Execution Roles**: CloudWatch Logs and VPC access
- **Cross-Account Roles**: Disaster recovery and backup access

Secrets Management:

- **AWS Secrets Manager**: Database credentials and API keys
- **Parameter Store**: Configuration values and non-sensitive settings
- **Rotation**: Automatic secret rotation with Lambda functions
- **Encryption**: KMS encryption for all stored secrets

9. Monitoring and Observability

CloudWatch Monitoring
Metrics Collection:

- **Infrastructure Metrics**: ECS, RDS, ElastiCache, and SQS standard metrics
- **Custom Metrics**: Business KPIs, tenant-specific metrics, and application performance
- **Log Aggregation**: Centralized logging from all services with retention policies
- **Performance Insights**: Database query performance and optimization recommendations

Alerting Strategy:

- **Critical Alerts**: Service outages, database failures, high error rates
- **Warning Alerts**: Resource utilization, performance degradation, queue backups
- **Business Alerts**: Tenant onboarding, billing issues, security events
- **Escalation**: PagerDuty integration with on-call rotation

Application Performance Monitoring
AWS X-Ray Integration:

- **Distributed Tracing**: Request flow across all services
- **Performance Analysis**: Bottleneck identification and optimization
- **Error Analysis**: Exception tracking with stack traces
- **Service Map**: Visual representation of service dependencies

Third-Party APM Options:

- **Datadog**: Comprehensive infrastructure and application monitoring
- **New Relic**: Deep application performance insights
- **Honeycomb**: Observability for complex distributed systems

10. Disaster Recovery and Business Continuity

Multi-Region Architecture
Primary Region: us-east-1 (N. Virginia) **Secondary Region**: us-west-2 (Oregon)
Cross-Region Components:

- **Aurora Global Database**: Sub-second data replication

- **S3 Cross-Region Replication**: Critical data backup
- **ECR Repository Replication**: Container image availability
- **Route 53 Health Checks**: Automatic DNS failover

Recovery Time Objectives:

- **RTO (Recovery Time Objective)**: 15 minutes for critical services
- **RPO (Recovery Point Objective)**: 1 minute data loss maximum
- **Automated Failover**: Health check-based automatic region switching
- **Manual Override**: Operations team manual failover capability

Backup Strategy

Database Backups:

- **Automated Backups**: Continuous backup with 7-day retention
- **Manual Snapshots**: Weekly snapshots with 30-day retention
- **Cross-Region Copies**: Daily backup replication to secondary region
- **Point-in-Time Recovery**: Restore to any point within retention window

Application State Backups:

- **ECS Task Definitions**: Version controlled with automated backup
- **Configuration Data**: Parameter Store and Secrets Manager replication
- **User Data**: S3 cross-region replication for critical user content
- **Infrastructure State**: Terraform state backup and versioning

11. Cost Optimization Strategy

Resource Optimization

Compute Cost Management:

- **Fargate Spot**: Development and non-critical workloads on Spot capacity
- **Right-Sizing**: Automated resource optimization based on utilization
- **Scheduled Scaling**: Scale down during low-usage periods
- **Reserved Capacity**: Reserved instances for predictable baseline capacity

Storage Cost Optimization:

- **S3 Intelligent Tiering**: Automatic storage class optimization
- **Lifecycle Policies**: Automated archival to reduce storage costs
- **Compression**: Reduce storage requirements for text-based content
- **CDN Caching**: Reduce origin requests and data transfer costs

Database Cost Management:

- **Aurora Serverless**: Pay-per-use scaling for variable workloads
- **Read Replica Optimization**: Scale read replicas based on actual load

- **Connection Pooling**: Reduce database instance requirements
- **Query Optimization**: Regular performance tuning to reduce resource usage

Cost Monitoring and Budgets

Budget Controls:

- **Service-Level Budgets**: Individual service spending limits
- **Environment Budgets**: Development, staging, and production budget separation
- **Alert Thresholds**: 50%, 80%, and 100% budget alerts
- **Automated Actions**: Scaling restrictions when approaching budget limits

Deployment and Operations Strategy

Infrastructure as Code

AWS CDK/CloudFormation:

- **Modular Stack Design**: Separate stacks for networking, compute, database, and monitoring
- **Environment Parity**: Identical infrastructure across development, staging, and production
- **Version Control**: Git-based infrastructure versioning with pull request reviews
- **Automated Testing**: Infrastructure validation and compliance testing

CI/CD Pipeline

Deployment Pipeline:

- **Source Control**: GitHub with branch protection and required reviews
- **Build Process**: Multi-stage Docker builds with security scanning
- **Testing**: Automated unit, integration, and end-to-end testing
- **Deployment**: Blue-green deployments with automated rollback

Release Strategy:

- **Feature Flags**: Gradual feature rollout with immediate disable capability
- **Canary Deployments**: 5% traffic routing for new deployments
- **Database Migrations**: Forward-compatible migrations with rollback plans
- **Health Validation**: Comprehensive health checks before traffic switch

Operational Procedures

Incident Management:

- **On-Call Rotation**: 24/7 coverage with escalation procedures
- **Runbooks**: Detailed troubleshooting guides for common issues
- **Post-Mortem Process**: Blameless post-incident reviews
- **Communication**: Automated status page updates and user notifications

Consequences

Positive Outcomes

Scalability and Performance:

- **Automatic Scaling**: Handles traffic spikes without manual intervention
- **Global Distribution**: Sub-100ms response times worldwide
- **High Availability**: 99.99% uptime with automatic failover
- **Performance Optimization**: CDN caching and edge computing for optimal user experience

Security and Compliance:

- **Defense in Depth**: Multiple security layers with managed security services
- **Compliance Ready**: Architecture supports SOC 2, GDPR, and other compliance requirements
- **Automated Security**: Security scanning and vulnerability management
- **Encryption Everywhere**: End-to-end encryption for data at rest and in transit

Operational Excellence:

- **Managed Services**: Reduced operational overhead with AWS managed services
- **Comprehensive Monitoring**: Full visibility into system performance and health
- **Automated Recovery**: Self-healing infrastructure with automatic failover
- **Cost Optimization**: Usage-based pricing with automatic resource optimization

Negative Consequences

Cost Considerations:

- **Premium Pricing**: Managed services and Fargate carry premium over self-managed alternatives
- **Data Transfer Costs**: Multi-AZ and cross-region data transfer charges
- **Reserved Capacity**: Requires accurate capacity planning for cost optimization
- **Monitoring Costs**: Comprehensive monitoring can become expensive at scale

Complexity and Dependencies:

- **Vendor Lock-in**: Heavy dependence on AWS services reduces portability
- **Service Integration**: Complex integration between multiple managed services
- **Debugging Complexity**: Distributed system debugging across multiple services
- **Learning Curve**: Team needs expertise in multiple AWS services

Operational Challenges:

- **Service Limits**: AWS service quotas may require limit increases
- **Regional Dependencies**: Feature availability varies by AWS region
- **Compliance Complexity**: Multi-service compliance requires comprehensive documentation
- **Disaster Recovery Testing**: Regular DR testing requires significant coordination

Risk Mitigation Strategies

Technical Risk Mitigation:

- **Multi-Region Deployment**: Reduces single-region failure risk
- **Circuit Breakers**: Prevent cascade failures between services
- **Graceful Degradation**: Service continues with reduced functionality during partial failures
- **Regular Load Testing**: Capacity validation before traffic spikes

Operational Risk Mitigation:

- **Comprehensive Documentation**: Detailed runbooks and troubleshooting guides
- **Training and Certification**: Team certification in relevant AWS services
- **Regular DR Drills**: Quarterly disaster recovery testing
- **Vendor Relationship**: AWS Enterprise Support with dedicated TAM

Cost Risk Mitigation:

- **Budget Monitoring**: Real-time cost tracking with automatic alerts
- **Resource Tagging**: Comprehensive tagging for cost allocation and optimization
- **Regular Reviews**: Monthly architecture reviews for cost optimization opportunities
- **Reserved Capacity Planning**: Annual capacity planning with reserved instance purchases

Tenant Architecture

Context

Taruvi Cloud requires a multi-tenant architecture that supports a hierarchical structure where users can create multiple sites, and each site can contain multiple applications. The platform must provide strong data isolation between sites while maintaining efficient resource utilization and operational simplicity.

Requirements

- **User Management**: Global user authentication and organization management
- **Site Isolation**: Complete data separation between different sites
- **Hierarchical Structure**: Users → Organizations → Sites → Apps
- **Scalability**: Support for thousands of sites with varying data volumes
- **Security**: Absolute data isolation between tenants
- **Performance**: Efficient database operations across all tenant levels
- **Compliance**: Support for data residency and regulatory requirements

Current Constraints

- Django-based application framework
- PostgreSQL as the primary database
- Need for real-time subscriptions and complex queries
- Multi-region deployment requirements
- Cost-effective scaling model

Decision

We will implement a **hybrid multi-tenant architecture** using Django-tenants with a three-tier schema structure:

Tier 1: Public Schema (Global)

The public schema contains platform-wide entities that are shared across all tenants:
Core Entities:

- **Users**: Global user accounts with authentication credentials
- **Organizations**: Business entities that can contain multiple users and sites
- **Organization Members**: User membership and roles within organizations
- **Sites**: Site definitions with metadata and routing information

- **Site Members**: User access permissions to specific sites
- **Billing**: Subscription plans, usage tracking, and payment information
- **Global Settings**: Platform-wide configuration and feature flags

Characteristics:

- Single shared schema for all platform users
- Handles authentication, authorization, and tenant discovery
- Manages billing and subscription logic
- Contains site routing and domain mapping
- Stores audit logs for compliance

Tier 2: Site Schema (Tenant)

Each site gets a dedicated schema providing complete isolation:

Site-Level Entities:

- **Apps**: Application definitions within the site
- **Site Configuration**: Site-specific settings and branding
- **Site Users**: Site-specific user profiles and preferences
- **Site Roles**: Custom roles defined at site level
- **Site Permissions**: Granular permissions for site resources
- **Site Audit Logs**: Site-specific activity tracking
- **Domain Configuration**: Custom domain and SSL settings

Characteristics:

- One schema per site with full PostgreSQL features
- Complete data isolation between sites
- Independent schema migrations per tenant
- Site-specific customization capabilities
- Isolated backup and recovery

Tier 3: App Data (Within Site Schema)

Application data resides within the site schema:

App-Level Entities:

- **Dynamic Tables**: User-defined schemas via JSON Schema
- **Function Definitions**: Serverless function metadata and code
- **MCP Server Configuration**: AI agent interaction settings
- **Integration Proxies**: External service connections
- **App-Specific Policies**: Access control and data governance

  Real-time Subscriptions: WebSocket connection management

- **File Storage Metadata**: Object storage references and permissions

Tenant Discovery and Routing

Site Identification Methods:

1. **Domain-Based**: `mysite.taruvi.cloud` or custom domains
2. **Header-Based**: `X-Tenant-ID` header for API requests
3. **Path-Based**: `/sites/{site-id}/` URL structure for admin interfaces

Routing Logic:

1. Incoming request hits the load balancer
2. Domain/header inspection determines the target site
3. Django-tenants middleware sets the schema context
4. All database operations automatically scope to the site schema
5. Cross-site operations require explicit public schema access

Data Flow Architecture

User Authentication Flow:

1. User authenticates against public schema
2. System retrieves accessible sites for the user
3. User selects target site or system auto-routes via domain
4. Request context switches to site-specific schema
5. Site-level permissions validated for requested operations

Cross-Schema Operations:

- **User Profile Sync**: Public user data synchronized to site schemas
- **Billing Aggregation**: Usage data flows from sites to public billing
- **Organization Management**: Centralized in public, referenced in sites
- **Audit Consolidation**: Site logs aggregated for organization-level reporting

Consequences

Positive

- **Strong Isolation**: Complete data separation between sites eliminates cross-tenant data leaks
- **Scalability**: Each site schema can be optimized independently
- **Customization**: Sites can have unique table structures and business logic
- **Performance**: Queries operate on smaller, focused datasets per site
- **Compliance**: Easy to implement data residency and retention policies per site
- **Backup/Recovery**: Granular control over data protection strategies

  Multi-Region: Sites can be distributed across different database instances

Negative

- **Complexity**: Managing multiple schemas increases operational overhead
- **Migration Challenges**: Schema changes must be applied across all tenants
- **Resource Overhead**: Each schema has PostgreSQL metadata overhead
- **Cross-Site Analytics**: Requires aggregation logic across multiple schemas
- **Connection Pooling**: More complex database connection management
- **Monitoring**: Need site-level monitoring and alerting systems

Mitigation Strategies

- **Automated Migration System**: Centralized schema version control and deployment
- **Resource Monitoring**: Per-site resource usage tracking and alerting
- **Connection Management**: Intelligent connection pooling with schema awareness
- **Backup Automation**: Automated backup scheduling for all tenant schemas
- **Performance Optimization**: Query optimization and indexing per tenant

Alternatives Considered

Alternative 1: Single Database with Row-Level Security (RLS)

Approach: Use PostgreSQL RLS policies to enforce tenant isolation within shared tables.
Pros: Simpler schema management, easier cross-tenant analytics **Cons**: Risk of policy misconfiguration, complex policy management, limited isolation
Rejection Reason: Insufficient isolation guarantees for enterprise customers and complex policy management overhead.

Alternative 2: Database-Per-Site

Approach: Create separate PostgreSQL databases for each site.
Pros: Maximum isolation, independent scaling **Cons**: High resource overhead, connection limit issues, operational complexity
Rejection Reason: Resource inefficiency and operational complexity outweigh benefits for expected site volumes.

Alternative 3: Microservices with Tenant-Aware Services

Approach: Split functionality into microservices with tenant routing.
Pros: Service-level isolation, independent scaling **Cons**: Network latency, complex service mesh, data consistency challenges
Rejection Reason: Premature architectural complexity for current scale requirements.

Alternative 4: Hybrid with Shared Core Tables

Approach: Mix of shared tables for common data and isolated tables for sensitive data.
Pros: Reduced redundancy, easier common operations **Cons**: Complex isolation logic, potential security risks
Rejection Reason: Inconsistent isolation model creates security vulnerabilities and architectural complexity.

Implementation Considerations

Schema Management

- Version-controlled schema definitions in Git
- Automated migration deployment across all tenant schemas
- Schema validation and rollback capabilities
- Performance impact assessment for schema changes

Security Model

- Public schema access restricted to platform operations
- Site schema access requires authenticated and authorized users
- Cross-schema operations use service accounts with limited permissions
- Regular security audits of tenant isolation

Monitoring and Observability

- Per-site resource utilization metrics
- Schema-level query performance monitoring
- Tenant-aware logging and alerting systems
- Data growth tracking and capacity planning

Disaster Recovery

- Site-level backup and restore capabilities
- Cross-region replication for high-value sites
- Point-in-time recovery with tenant granularity
- Automated failover procedures

Authentication systems

Hybrid Access Control Model for Taruvi

Context

Taruvi is an AI backend as a service platform that enables frontend agents to create production applications. Our platform faces unique access control challenges that traditional single-model approaches cannot adequately address:

Business Requirements

- **Multi-tenant SaaS Architecture**: Multiple organizations with distinct access patterns
- **Dynamic AI Agent Interactions**: Frontend agents need context-aware permissions that change based on data sensitivity, processing requirements, and organizational policies
- **Production-Ready Applications**: End applications built on Taruvi must support enterprise-grade access control
- **Scalable Permission Management**: Support for thousands of users across hundreds of organizations with varying access patterns
- **Real-time Decision Making**: AI operations require sub-second authorization decisions
- **Compliance Requirements**: GDPR, SOX, and industry-specific regulations demand fine-grained access control

Technical Challenges

1. **Role Explosion Problem**: Pure RBAC would require hundreds of roles to cover all organizational structures and AI operation types
2. **Context-Sensitive Operations**: AI operations often depend on data classification, processing location, time constraints, and resource availability
3. **Dynamic Resource Creation**: AI agents create resources dynamically, requiring flexible permission inheritance
4. **Cross-Tenant Scenarios**: Some AI operations need controlled access across organizational boundaries
5. **Audit and Compliance**: Need detailed logs of who accessed what, when, and under what conditions

Current Pain Points

- Existing RBAC systems are too rigid for AI agent workflows
- Pure ABAC systems are too complex for standard user management
- Django's built-in permissions are insufficient for multi-tenant, context-aware scenarios
- Third-party solutions don't understand AI-specific access patterns

Decision

We will implement a **Hybrid Access Control Model** combining Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) in Django, with the following architecture:

Core Components

1. RBAC Foundation Layer

- **Organizational Roles**: Standard business roles (Admin, Developer, Data Scientist, Viewer)
- **System Roles**: Platform-specific roles (AI Agent Creator, Model Trainer, API Publisher)
- **Tenant-Specific Roles**: Customizable roles per organization
- **Permission Hierarchies**: Role inheritance and permission aggregation

2. ABAC Enhancement Layer

- **User Attributes**: Department, security clearance, location, device trust level
- **Resource Attributes**: Data classification, processing requirements, tenant ownership
- **Environmental Attributes**: Time, location, network context, request origin
- **Action Attributes**: Operation type, data sensitivity level, processing scope

3. Policy Engine Integration

- **Django-OPA Bridge**: Integration with Open Policy Agent for complex policy evaluation
- **Rule Cache**: Redis-backed caching for frequently evaluated policies
- **Policy Versioning**: Git-based policy management with rollback capabilities
- **Dynamic Policy Loading**: Hot-reload capabilities for policy updates

Implementation Strategy

Phase 1: RBAC Foundation

- **Django Custom User Model**: Extended user model with tenant relationships
- **Role Management System**: Hierarchical role definitions with inheritance
- **Permission Framework**: Django's permission system enhanced for multi-tenancy
- **Tenant Isolation**: Row-level security ensuring cross-tenant data protection

Phase 2: Attribute Integration

- **User Attribute Store**: JSON fields and related models for user attributes
- **Resource Tagging System**: Comprehensive metadata system for all platform resources
- **Context Collection**: Middleware for gathering environmental attributes
- **Attribute Synchronization**: Integration with HR systems, identity providers

Phase 3: Policy Engine

- **OPA Integration**: Policy evaluation service with Django integration

  Policy Definition Language: Domain-specific language for Taruvi access policies

- **Policy Testing Framework**: Unit and integration testing for access policies
- **Policy Analytics**: Monitoring and analysis of policy effectiveness

Django-Specific Implementation

Custom Decorators and Middleware

- **@requires_permission**: Enhanced decorator combining role and attribute checks
- **AttributeMiddleware**: Automatic attribute collection and context building
- **PolicyEvaluationMiddleware**: Real-time policy evaluation for all requests

Database Design

- **Multi-tenant Architecture**: Tenant-aware models with proper isolation
- **Attribute Storage**: Flexible schema for storing diverse attribute types
- **Audit Logging**: Comprehensive logging of all access decisions
- **Policy Cache Tables**: Database-backed policy caching for performance

Integration Points

- **Django Admin Integration**: Custom admin interfaces for role and policy management
- **DRF Integration**: Custom permission classes for API endpoints
- **Celery Task Security**: Attribute-based task execution permissions
- **WebSocket Security**: Real-time connection authorization

Consequences

Positive Outcomes

Security Benefits

- **Fine-grained Control**: Precise access control matching business requirements
- **Context Awareness**: Decisions based on complete request context
- **Scalability**: Handles complex organizational structures without role explosion
- **Compliance Ready**: Detailed audit trails and policy-driven access control

Operational Benefits

- **Flexibility**: Easy adaptation to changing business requirements
- **Maintainability**: Clear separation between roles and contextual policies
- **Performance**: Cached policies ensure fast authorization decisions
- **Debuggability**: Clear audit trails for troubleshooting access issues

Business Benefits

- **Faster Onboarding**: Attribute-based user provisioning
- **Reduced Support Overhead**: Self-explanatory permission denials
- **Enterprise Sales**: Enterprise-grade access control as a selling point

- **Compliance Acceleration**: Built-in compliance framework

Challenges and Mitigations

Complexity Management

- **Challenge**: Hybrid systems are more complex than single-model approaches
- **Mitigation**: Comprehensive documentation, training programs, and gradual rollout

Performance Considerations

- **Challenge**: Attribute evaluation can impact response times
- **Mitigation**: Multi-layer caching strategy, policy optimization, and performance monitoring

Policy Governance

- **Challenge**: Policy sprawl and conflicting rules
- **Mitigation**: Policy review processes, automated testing, and governance frameworks

Migration Path

- **Challenge**: Transitioning existing simple permissions to hybrid model
- **Mitigation**: Backward compatibility layer and gradual migration tools

Monitoring and Success Metrics

- **Authorization Decision Latency**: Target <50ms for 95th percentile
- **Policy Cache Hit Rate**: Target >90% cache hit rate
- **Security Incident Reduction**: Measure reduction in access-related security incidents
- **User Onboarding Time**: Track time to grant appropriate access to new users
- **Policy Effectiveness**: Monitor false positives/negatives in access decisions

Alternative Approaches Considered

Pure RBAC Approach

- **Rejected**: Would require 200+ roles for our use cases, leading to management complexity
- **Insufficient**: Cannot handle context-dependent AI operations

Pure ABAC Approach

- **Rejected**: Too complex for standard user management scenarios
- **Risky**: Higher implementation risk and maintenance overhead

Third-Party Solutions

- **Rejected**: Existing solutions don't understand AI workflow patterns
- **Cost**: Prohibitive licensing costs for our scale

Implementation Timeline

- **Phase 1 (Months 1-3)**: RBAC foundation and Django integration

- **Phase 2 (Months 4-6)**: Attribute collection and basic ABAC policies
- **Phase 3 (Months 7-9)**: Full policy engine integration and optimization
- **Phase 4 (Months 10-12)**: Advanced features and enterprise integrations

Success Criteria

1. **Security**: Zero unauthorized access incidents in production
2. **Performance**: Authorization decisions under 50ms latency
3. **Usability**: 90% user satisfaction with permission system clarity
4. **Compliance**: Pass all relevant security audits
5. **Scalability**: Support 10,000+ users across 100+ tenants without performance degradation

This hybrid approach positions Taruvi as an enterprise-ready platform while maintaining the flexibility needed for AI-driven applications and dynamic resource management.

Untitled

Data Service - App Data Storage

Context

Taruvi Cloud requires a flexible data storage solution that can accommodate diverse AI agent workflows while maintaining strict access control and providing developer-friendly APIs similar to Supabase. The platform must support both structured relational data and flexible document-style storage, with consistent API interfaces and security enforcement across both paradigms.

Requirements

- **API Compatibility**: Supabase-style REST and GraphQL APIs with real-time subscriptions
- **Schema Flexibility**: Support for both rigid relational schemas and flexible document structures
- **Access Control**: Mandatory RBAC+ABAC enforcement at the data layer
- **Performance**: Efficient querying for both storage types with optimized indexing
- **Developer Experience**: Consistent API regardless of underlying storage type
- **Schema Evolution**: Support for schema changes without data migration
- **Multi-App Isolation**: Apps within a site can choose different storage strategies
- **AI Agent Integration**: MCP server compatibility for programmatic schema management

Current Constraints

- PostgreSQL as the primary database with JSONB support
- Django ORM limitations for dynamic schema generation
- Need for real-time data synchronization
- Complex access control requirements with dynamic filtering
- Site-level tenant isolation already established
- Performance requirements for complex JSONB queries with indexing

Decision

We will implement a **Dual Storage Architecture** with unified API access and mandatory access control enforcement, supporting two distinct storage paradigms within each site schema.

Storage Architecture Overview

Core Site Schema Tables

Every site schema contains these foundational tables:

Schema Management:

- **app_schemas**: Pydantic JSON Schema definitions for each app's data models
- **app_configurations**: App-level settings including storage type selection
- **schema_versions**: Version history and migration tracking for schema changes
- **validation_rules**: Custom validation logic beyond JSON Schema constraints

Access Control:

- **roles**: Site-specific role definitions with hierarchical inheritance
- **permissions**: Granular permission definitions for resources and operations
- **policies**: ABAC policy rules with condition expressions
- **role_assignments**: User-to-role mappings with optional time-based constraints
- **resource_filters**: Dynamic row-level security filters per role/resource combination

API Management:

- **api_tokens**: Scoped authentication tokens with permission subsets
- **rate_limits**: Configurable throttling rules per endpoint/user/role
- **audit_logs**: Comprehensive access logging for compliance and debugging
- **webhook_subscriptions**: Real-time event notification configurations

Storage Type 1: JSONB Store
Architecture:

- **Separate table per entity type**: `app_{app_id}_{entity_type}_jsonb`
- **Entity-specific indexing**: Dedicated indexes per entity type for optimal query performance
- **Polymorphic support**: Single entity type can have variant schemas through versioning
- **Type-specific optimization**: Each table optimized for its specific JSONB field patterns

Table Structure per Entity Type:

```
app_{app_id}_{entity_type}_jsonb:
- id: UUID (Primary Key)
- data: JSONB (Full entity data)
- created_at: TIMESTAMP- updated_at: TIMESTAMP- version: INTEGER- created_by: UUID (Foreign Key to users)
- metadata: JSONB (System metadata, tags, etc.)
- search_vector: TSVECTOR (Full-text search index)
- computed_fields: JSONB (Derived/calculated values)
```

Entity-Specific Indexing Strategy:

- **Field-Specific GIN Indexes**: `CREATE INDEX ON app_1_users_jsonb USING GIN ((data->'email'))`
- **Composite JSONB Indexes**: `CREATE INDEX ON app_1_orders_jsonb USING GIN ((data->'status'), (data->'customer_id'))`
- **Partial Indexes**: `CREATE INDEX ON app_1_products_jsonb ((data->'price')) WHERE (data->>'active')::boolean = true`

- **Expression Indexes**: `CREATE INDEX ON app_1_events_jsonb ((data->>'timestamp')::timestamp) WHERE data->>'type' = 'user_action'`
- **Full-Text Indexes**: `CREATE INDEX ON app_1_posts_jsonb USING GIN (search_vector)`

Advanced JSONB Features per Entity:

- **Computed Columns**: Virtual columns derived from JSONB data for frequently queried values
- **Constraint Validation**: CHECK constraints on JSONB fields using PostgreSQL JSON functions
- **Trigger-Based Indexing**: Automatic maintenance of search vectors and computed fields
- **Schema Evolution Tracking**: Per-entity schema version tracking within the JSONB structure

Advantages:

- **Optimized Indexing**: Entity-specific indexes dramatically improve query performance
- **Schema Flexibility**: No database migrations required for schema changes
- **Type-Specific Optimization**: Each entity type can have specialized indexing strategies
- **Rapid Prototyping**: Instant schema deployment and modification
- **Version Management**: Built-in entity versioning without schema changes
- **Complex Data Types**: Native support for nested objects and arrays
- **Query Performance**: Near-relational performance for indexed JSONB fields

Trade-offs:

- **Table Proliferation**: More tables to manage compared to single-table approach
- **Cross-Entity Queries**: Joins across entity types require explicit UNION operations
- **Storage Overhead**: JSON structure adds storage cost per entity type
- **Index Maintenance**: Multiple GIN indexes require more maintenance overhead

Storage Type 2: Flat Tables

Architecture:

- Dedicated table per entity type: `app_{app_id}_{entity_type}`
- Dynamic DDL generation from Pydantic JSON Schema
- Traditional relational constraints and indexes
- Foreign key relationships enforced at database level

Schema Generation Process:

1. Pydantic JSON Schema parsed for field definitions
2. PostgreSQL DDL generated with appropriate column types
3. Indexes created based on schema metadata hints
4. Foreign key constraints established for relationships
5. Check constraints applied for validation rules

Advantages:

- **Query Performance**: Optimal indexing and query optimization
- **Data Integrity**: Database-level constraints and foreign keys
- **Storage Efficiency**: Minimal storage overhead
- **SQL Compatibility**: Standard SQL operations and joins
- **Analytics Friendly**: Direct compatibility with BI tools

Trade-offs:

- **Schema Rigidity**: Database migrations required for schema changes
- **Deployment Complexity**: DDL operations during schema updates
- **Relationship Limits**: Complex nested relationships challenging to model
- **Migration Risk**: Data migration required for breaking schema changes

Unified API Layer

REST API Design

Endpoint Structure:

- `GET /api/v1/sites/{site_id}/apps/{app_id}/{entity_type}`
- `POST /api/v1/sites/{site_id}/apps/{app_id}/{entity_type}`
- `PUT /api/v1/sites/{site_id}/apps/{app_id}/{entity_type}/{id}`
- `DELETE /api/v1/sites/{site_id}/apps/{app_id}/{entity_type}/{id}`

Query Parameters:

- **Filtering**: `?filter={"field": "value", "age": {"$gt": 18}}`
- **Sorting**: `?sort=[{"field": "created_at", "direction": "desc"}]`
- **Pagination**: `?page=1&limit=50` or `?offset=0&limit=50`
- **Field Selection**: `?select=["id", "name", "email"]`
- **Relationships**: `?expand=["profile", "orders.items"]`

GraphQL Schema

Dynamic Schema Generation:

- GraphQL schema auto-generated from Pydantic JSON Schema definitions
- Consistent query interface regardless of storage type
- Real-time subscriptions for data changes
- Automatic relationship resolution for both storage types

Query Capabilities:

- **Nested Queries**: Deep relationship traversal
- **Filtered Subscriptions**: Real-time updates with access control
- **Aggregations**: Count, sum, average operations

  Mutations: CRUD operations with validation

Enhanced Query Performance for JSONB Store

Index Management System

Automatic Index Creation:

- **Schema Analysis**: Automatic detection of frequently queried JSONB paths
- **Usage Pattern Detection**: Query log analysis to identify optimal index candidates
- **Index Recommendation Engine**: AI-driven suggestions for new indexes based on query patterns
- **Dynamic Index Creation**: Background index creation during low-traffic periods

Entity-Specific Optimization:

- **User Entities**: Indexes on email, username, profile data for authentication queries
- **Content Entities**: Full-text search indexes on title, description, content fields
- **Transaction Entities**: Composite indexes on amount, currency, timestamp for financial queries
- **Event Entities**: Time-series indexes optimized for chronological queries
- **Relationship Entities**: Specialized indexes for foreign key-like JSONB references

Cross-Entity Query Support

Union-Based Queries:

- **Federated Search**: Search across multiple entity types using UNION ALL
- **Cross-Entity Aggregations**: Combined statistics across related entity types
- **Polymorphic Queries**: Single API endpoint returning multiple entity types
- **Performance Optimization**: Parallel execution of cross-entity queries

Relationship Simulation:

- **Reference Resolution**: Automatic resolution of JSONB references to other entities
- **Denormalization Support**: Optional data duplication for performance-critical queries
- **Consistency Management**: Background jobs to maintain referential consistency
- **Graph-Like Queries**: Deep relationship traversal across JSONB entity tables

Access Control Enforcement

RBAC + ABAC Hybrid Model

Role-Based Access Control:

- **Hierarchical Roles**: Support for role inheritance and composition
- **Permission Scoping**: Permissions scoped to resources, operations, and conditions
- **Dynamic Assignment**: Time-based and condition-based role assignments
- **Delegation**: Users can delegate subset of permissions to others

Attribute-Based Access Control:

- **Context Attributes**: User attributes, resource attributes, environmental factors
- **Policy Rules**: Condition expressions using attribute values

- **Dynamic Evaluation**: Real-time policy evaluation during request processing
- **Policy Inheritance**: Policies can inherit and override parent policies

Access Control Implementation

Request Processing Flow:

1. **Authentication**: Token validation and user identification
2. **Authorization**: Role and permission verification for requested operation
3. **Policy Evaluation**: ABAC policy rules evaluated against request context
4. **Filter Generation**: Dynamic WHERE clauses generated based on access rules
5. **Query Execution**: Original query modified with access control filters
6. **Result Filtering**: Post-query filtering for complex access rules
7. **Response Sanitization**: Field-level access control applied to response

Storage-Agnostic Filtering:

- **JSONB Store**: JSONB path operations for filtering nested data across entity tables
- **Flat Tables**: Standard SQL WHERE clauses with JOINs
- **Consistent Interface**: Same filter syntax works for both storage types
- **Performance Optimization**: Index utilization for common filter patterns

Security Enforcement Points

Database Level:

- **Row-Level Security**: PostgreSQL RLS policies as backup enforcement per entity table
- **Column Security**: Sensitive field masking based on permissions
- **Audit Triggers**: Automatic logging of all data access and modifications
- **Connection Security**: Schema-scoped database connections per site

Application Level:

- **Request Validation**: Schema validation before database operations
- **Response Filtering**: Field-level access control on response data
- **Rate Limiting**: Per-user, per-role, and per-resource throttling
- **Audit Logging**: Comprehensive request/response logging

Schema Management System

Pydantic JSON Schema Integration

Schema Definition:

- Standard JSON Schema format with Pydantic extensions
- Custom metadata for indexing, validation, and relationship hints
- Version control with backward compatibility tracking
- Validation rule definitions with custom Python expressions

Entity-Specific Schema Features:

- **Index Hints**: Schema annotations for optimal index creation
- **Search Configuration**: Full-text search field definitions
- **Computed Field Definitions**: Derived field calculations from base JSONB data
- **Relationship Mapping**: Foreign key-like references between JSONB entities

Migration Strategy

JSONB Store Migrations:

- **Schema Updates**: No database changes required for most updates
- **Index Management**: Automatic index creation/deletion based on schema changes
- **Entity Table Creation**: New entity types trigger automatic table creation
- **Data Transformation**: Background jobs for computed field updates
- **Validation Updates**: New validation rules applied to future writes
- **Backward Compatibility**: Multiple schema versions supported simultaneously

Flat Table Migrations:

- **Additive Changes**: New columns added without downtime
- **Breaking Changes**: Blue-green deployment strategy for table recreation
- **Data Migration**: Automatic migration scripts generated from schema diff
- **Rollback Support**: Schema version rollback with data transformation

Performance Considerations

Query Optimization

JSONB Store Optimization:

- **Entity-Specific GIN Indexes**: Automatic index creation for frequently queried JSONB paths
- **Partial Indexes**: Selective indexing based on entity-specific query patterns
- **Expression Indexes**: Computed indexes for derived values per entity type
- **Query Planning**: JSONB query optimization based on access patterns per entity
- **Parallel Query Execution**: Cross-entity queries executed in parallel

Flat Table Optimization:

- **Automatic Indexing**: Index generation from schema metadata
- **Composite Indexes**: Multi-column indexes for complex queries
- **Foreign Key Indexes**: Automatic indexing of relationship columns
- **Query Analysis**: Performance monitoring and optimization suggestions

Caching Strategy

Schema Caching:

- **In-Memory Schema Cache**: Parsed schemas cached in application memory

- **Redis Schema Store**: Distributed schema cache across application instances
- **CDN Distribution**: Schema definitions cached at edge locations
- **Invalidation Strategy**: Automatic cache invalidation on schema updates

Data Caching:

- **Entity-Specific Caching**: Separate cache strategies per entity type
- **Query Result Caching**: Configurable caching for expensive cross-entity queries
- **Individual Entity Caching**: Per-entity caching with TTL
- **Relationship Caching**: Pre-computed relationship data across JSONB entities
- **Real-time Invalidation**: Cache invalidation on data changes

Consequences

Positive Outcomes

Developer Experience:

- **Flexible Architecture**: Choose optimal storage type per use case
- **Consistent APIs**: Same interface regardless of storage choice
- **Rapid Development**: Quick schema iteration and deployment
- **Type Safety**: Full TypeScript/Python type generation from schemas

Performance Benefits:

- **Optimized JSONB Queries**: Entity-specific indexing dramatically improves performance
- **Selective Index Management**: Only relevant indexes created per entity type
- **Parallel Processing**: Cross-entity queries can be parallelized effectively
- **Storage Efficiency**: Specialized storage patterns per entity type

Security Benefits:

- **Mandatory Access Control**: Impossible to bypass security enforcement
- **Fine-Grained Control**: Resource and field-level access control
- **Audit Compliance**: Comprehensive audit trail for all data access
- **Policy Flexibility**: Complex access rules without custom code

Negative Consequences

Complexity Overhead:

- **Table Proliferation**: More database objects to manage in JSONB store
- **Index Management**: Complex index maintenance across entity-specific tables
- **Cross-Entity Complexity**: More complex queries when joining across entity types
- **Monitoring Overhead**: Need to monitor performance across multiple entity tables

Operational Challenges:

- **Schema Validation**: More complex validation across entity-specific tables
- **Migration Complexity**: Managing schema changes across multiple entity tables
- **Backup Strategies**: Storage-type-specific backup across many tables
- **Query Analysis**: Complex performance analysis across entity-distributed data

Performance Trade-offs:

- **Cross-Entity Queries**: UNION operations may be slower than single-table scans
- **Memory Usage**: More indexes mean higher memory utilization
- **Storage Overhead**: Table overhead multiplied by number of entity types
- **Connection Pooling**: More tables require careful connection management

Risk Mitigation

Performance Risks:

- **Query Optimization**: Automated cross-entity query optimization
- **Index Monitoring**: Real-time index usage and performance monitoring
- **Capacity Planning**: Entity-specific growth prediction and scaling
- **Performance Testing**: Load testing across all entity table combinations

Operational Risks:

- **Table Management**: Automated table lifecycle management
- **Index Maintenance**: Automated index optimization and cleanup
- **Schema Consistency**: Cross-entity schema validation and consistency checks
- **Migration Safety**: Comprehensive testing of entity-specific migrations

Alternatives Considered

Alternative 1: Single JSONB Table per App

Approach: Store all entity types in single table with type discrimination.
Pros: Simpler table management, easier cross-entity queries **Cons**: Poor indexing performance, generic indexes less efficient, harder to optimize per entity
Rejection Reason: Significant performance degradation for entity-specific queries and inability to optimize indexes per entity type.

Alternative 2: Hybrid JSONB with Extracted Columns

Approach: JSONB storage with frequently queried fields extracted to regular columns.
Pros: Best of both worlds, optimized common queries **Cons**: Complex schema management, dual maintenance overhead
Rejection Reason: Adds complexity without providing the full flexibility of pure JSONB or performance of flat tables.

Alternative 3: Document Database Integration

Approach: Use MongoDB or similar for JSONB-style storage alongside PostgreSQL.

Pros: Purpose-built for document storage, excellent indexing **Cons**: Operational complexity, data consistency challenges, dual security models

Rejection Reason: Operational overhead and transaction complexity across two database systems.

This revised architecture provides optimal indexing performance for JSONB storage while maintaining the flexibility benefits of document-style data storage, ensuring both storage types can achieve excellent query performance through entity-specific optimization strategies.

Environment Variables and Management

Context

Taruvi Cloud requires a comprehensive environment variables and secrets management system that supports the hierarchical tenant architecture. The system must provide secure storage, controlled access, and inheritance patterns between site-level and app-level configurations while maintaining strong security boundaries and audit capabilities.

Requirements

- **Hierarchical Configuration**: Site-level variables that cascade to app-level with override capabilities
- **Security Classification**: Clear distinction between public configuration and sensitive secrets
- **Access Control**: Integration with existing RBAC/ABAC system for fine-grained permissions
- **API Access**: RESTful and GraphQL APIs for programmatic access to variables and secrets
- **Encryption**: Strong encryption for sensitive data at rest and in transit
- **Audit Trail**: Comprehensive logging of all secret access and modifications
- **Performance**: Fast retrieval for high-frequency API calls
- **Secret Rotation**: Support for automated secret rotation and versioning
- **Integration**: Compatibility with external secret management systems
- **Site-as-Environment**: Each site represents a complete environment (dev/staging/prod sites are separate)

Current Constraints

- Multi-tenant architecture with site-based schema isolation
- PostgreSQL as primary data store
- Django-based application with existing RBAC system
- High-frequency API access requirements
- Compliance requirements for audit logging and encryption
- Sites represent complete environments rather than having environment subdivision

Decision

We will implement a **Hierarchical Environment Management System** using **django-encrypted-model-fields** for secure field-level encryption, with site-to-app inheritance cascading and comprehensive access control integration.

Architecture Overview

Storage Strategy

Two-Tier Variable System:

- **Site-Level Variables**: Global configuration shared across all apps within a site
- **App-Level Variables**: Application-specific configuration with inheritance from site level
- **Variable Types**: Clear separation between public configuration and encrypted secrets
- **Site-as-Environment**: Each site represents a complete environment context

Encryption Solution: django-encrypted-model-fields

Package Selection: We will use `django-encrypted-model-fields` which provides:

- **Fernet Encryption**: Symmetric encryption using the cryptography library's Fernet implementation
- **Automatic Key Management**: Keys managed through Django settings with rotation support
- **Transparent Field Encryption**: Encrypted fields work seamlessly with Django ORM
- **Battle-Tested**: Widely used in production Django applications
- **Active Maintenance**: Well-maintained with regular security updates

Key Management:

- **DJANGO_FERNET_KEYS**: List of Fernet keys in Django settings
- **Key Rotation**: Multiple keys supported for seamless rotation
- **Environment Variables**: Keys stored as environment variables, not in code
- **Per-Tenant Keys**: Different key sets per tenant using tenant-aware settings

Database Schema Design

Site Schema Tables:

```
python
Run CodeCopy code
# models.pyfrom encrypted_model_fields.fields import EncryptedTextField
from django.db import models
import uuid

class SiteEnvironmentVariable(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    # site_id is implicit via tenant schema
    key = models.CharField(max_length=255, db_index=True)

    # Conditional encryption based on variable_type
    value = models.TextField()  # For config variables
    encrypted_value = EncryptedTextField(null=True, blank=True)  # For secrets

    variable_type = models.CharField(
        max_length=20,
        choices=[('config', 'Configuration'), ('secret', 'Secret')],
        default='config'
    )
    description = models.TextField(blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    created_by = models.UUIDField()  # Reference to user
```

```
    last_accessed_at = models.DateTimeField(null=True, blank=True)
    access_count = models.PositiveIntegerField(default=0)
    version = models.PositiveIntegerField(default=1)
    metadata = models.JSONField(default=dict, blank=True)

    class Meta:
        unique_together = [['key']]  # Unique key per site
        indexes = [
            models.Index(fields=['key']),
            models.Index(fields=['variable_type']),
            models.Index(fields=['created_at']),
        ]

    def get_value(self):
        """Get decrypted value regardless of type"""if self.variable_type == 'secret':
            return self.encrypted_value
        return self.value

    def set_value(self, new_value):
        """Set value with appropriate encryption"""if self.variable_type == 'secret':
            self.encrypted_value = new_value
            self.value = ''  # Clear plaintextelse:
            self.value = new_value
            self.encrypted_value = Noneclass AppEnvironmentVariable(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    app_id = models.UUIDField(db_index=True)  # Reference to app
    key = models.CharField(max_length=255, db_index=True)

    # Conditional encryption based on variable_type
    value = models.TextField()  # For config variables
    encrypted_value = EncryptedTextField(null=True, blank=True)  # For secrets

    variable_type = models.CharField(
        max_length=20,
        choices=[('config', 'Configuration'), ('secret', 'Secret')],
        default='config'
    )
    description = models.TextField(blank=True)
    overrides_site = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    created_by = models.UUIDField()
    last_accessed_at = models.DateTimeField(null=True, blank=True)
    access_count = models.PositiveIntegerField(default=0)
    version = models.PositiveIntegerField(default=1)
    metadata = models.JSONField(default=dict, blank=True)

    class Meta:
        unique_together = [['app_id', 'key']]  # Unique key per app
        indexes = [
            models.Index(fields=['app_id', 'key']),
            models.Index(fields=['app_id', 'variable_type']),
            models.Index(fields=['created_at']),
        ]

    def get_value(self):
        """Get decrypted value regardless of type"""if self.variable_type == 'secret':
```

```
            return self.encrypted_value
        return self.value

    def set_value(self, new_value):
        """Set value with appropriate encryption"""if self.variable_type == 'secret':
            self.encrypted_value = new_value
            self.value = ''  # Clear plaintextelse:
            self.value = new_value
            self.encrypted_value = Noneclass EnvironmentVariablePermission(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    variable_id = models.UUIDField()
    variable_scope = models.CharField(
        max_length=10,
        choices=[('site', 'Site'), ('app', 'App')]
    )
    role_id = models.UUIDField()
    permission_type = models.CharField(
        max_length=20,
        choices=[
            ('read', 'Read'),
            ('write', 'Write'),
            ('delete', 'Delete'),
            ('rotate', 'Rotate')
        ]
    )
    created_at = models.DateTimeField(auto_now_add=True)
    created_by = models.UUIDField()


class SecretRotationHistory(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    variable_id = models.UUIDField()
    variable_scope = models.CharField(
        max_length=10,
        choices=[('site', 'Site'), ('app', 'App')]
    )
    old_version = models.PositiveIntegerField()
    new_version = models.PositiveIntegerField()
    rotation_type = models.CharField(
        max_length=20,
        choices=[
            ('manual', 'Manual'),
            ('scheduled', 'Scheduled'),
            ('emergency', 'Emergency')
        ]
    )
    rotated_at = models.DateTimeField(auto_now_add=True)
    rotated_by = models.UUIDField()
    reason = models.TextField(blank=True)


class VariableAccessLog(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    variable_id = models.UUIDField()
    variable_scope = models.CharField(
        max_length=10,
        choices=[('site', 'Site'), ('app', 'App')]
```

```
    )
    variable_key = models.CharField(max_length=255, db_index=True)
    user_id = models.UUIDField()
    access_type = models.CharField(
        max_length=20,
        choices=[
            ('read', 'Read'),
            ('write', 'Write'),
            ('delete', 'Delete'),
            ('rotate', 'Rotate')
        ]
    )
    client_ip = models.GenericIPAddressField()
    user_agent = models.TextField()
    access_time = models.DateTimeField(auto_now_add=True)
    response_status = models.PositiveIntegerField()
    error_message = models.TextField(blank=True)

    class Meta:
        indexes = [
            models.Index(fields=['variable_key', 'access_time']),
            models.Index(fields=['user_id', 'access_time']),
            models.Index(fields=['access_type', 'access_time']),
        ]
```

Variable Inheritance and Resolution

Inheritance Service Implementation

```
python
Run CodeCopy code
# services/environment_service.pyfrom typing import Dict, Any, Optionalfrom django.core.cache import cache

class EnvironmentVariableService:

    def resolve_variables(
        self,
        app_id: str,
        user_id: str) -> Dict[str, Any]:
        """
        Resolve all variables for an app with inheritance from site level
        """
        cache_key = f"resolved_vars:{app_id}:{user_id}"
        cached = cache.get(cache_key)
        if cached:
            return cached

        # Get site-level variables
        site_vars = self._get_site_variables(user_id)

        # Get app-level variables
        app_vars = self._get_app_variables(app_id, user_id)

        # Apply inheritance and override rules
        resolved = self._apply_inheritance(site_vars, app_vars)

        # Cache for 5 minutes (shorter for secrets)
```

```
        cache.set(cache_key, resolved, 300)

        return resolved

    def _get_site_variables(self, user_id: str) -> Dict:
        """Get site variables user has permission to read"""from .models import SiteEnvironmentVariable

        variables = SiteEnvironmentVariable.objects.all().select_related()

        result = {}
        for var in variables:
            if self._check_read_permission(var, user_id, 'site'):
                result[var.key] = {
                    'value': var.get_value(),
                    'variable_type': var.variable_type,
                    'description': var.description,
                    'metadata': var.metadata
                }
                # Log access
                self._log_access(var, user_id, 'read', 'site')

        return result

    def _get_app_variables(self, app_id: str, user_id: str) -> Dict:
        """Get app variables user has permission to read"""from .models import AppEnvironmentVariable

        variables = AppEnvironmentVariable.objects.filter(
            app_id=app_id
        ).select_related()

        result = {}
        for var in variables:
            if self._check_read_permission(var, user_id, 'app'):
                result[var.key] = {
                    'value': var.get_value(),
                    'variable_type': var.variable_type,
                    'description': var.description,
                    'metadata': var.metadata,
                    'overrides_site': var.overrides_site
                }
                # Log access
                self._log_access(var, user_id, 'read', 'app')

        return result

    def _apply_inheritance(self, site_vars: Dict, app_vars: Dict) -> Dict:
        """Apply inheritance rules to resolve final variable set"""
        resolved = {}

        # Start with site variablesfor key, var_data in site_vars.items():
            resolved[key] = {
                **var_data,
                'source': 'site',
                'inherited': True
            }

        # Apply app overrides and additionsfor key, var_data in app_vars.items():
```

```
            if var_data.get('overrides_site', False) and key in resolved:
                # App explicitly overrides site variable
                resolved[key] = {
                    **var_data,
                    'source': 'app',
                    'overridden': True
                }
            elif key not in resolved:
                # App-specific variable
                resolved[key] = {
                    **var_data,
                    'source': 'app',
                    'app_specific': True
                }
            # If app variable exists but doesn't override, site takes precedencereturn resolved

    def _check_read_permission(self, variable, user_id: str, scope: str) -> bool:
        """Check if user has read permission for variable"""# Implementation would check RBAC/ABAC permissions# This is a placeholder for the actual permission checking logicreturn Truedef _log_access(self, variable, user_id: str, access_type: str, scope: str):
        """Log variable access for audit trail"""from .models import VariableAccessLog
        # Implementation would create audit log entrypass
```

API Design

REST API Endpoints

Site-Level Variables:

```
bash
Copy code
GET    /api/v1/sites/{site_id}/variables
POST   /api/v1/sites/{site_id}/variables
GET    /api/v1/sites/{site_id}/variables/{key}
PUT    /api/v1/sites/{site_id}/variables/{key}
DELETE /api/v1/sites/{site_id}/variables/{key}
POST   /api/v1/sites/{site_id}/variables/{key}/rotate
```

App-Level Variables:

```
bash
Copy code
GET    /api/v1/sites/{site_id}/apps/{app_id}/variables
POST   /api/v1/sites/{site_id}/apps/{app_id}/variables
GET    /api/v1/sites/{site_id}/apps/{app_id}/variables/{key}
PUT    /api/v1/sites/{site_id}/apps/{app_id}/variables/{key}
DELETE /api/v1/sites/{site_id}/apps/{app_id}/variables/{key}
POST   /api/v1/sites/{site_id}/apps/{app_id}/variables/{key}/rotate
```

Resolved Variables (Inheritance Applied):

```
bash
Copy code
GET    /api/v1/sites/{site_id}/apps/{app_id}/variables/resolved
GET    /api/v1/sites/{site_id}/apps/{app_id}/variables/resolved/{key}
```

Audit and Management:

```
bash
Copy code
GET    /api/v1/sites/{site_id}/variables/audit
```

```
GET    /api/v1/sites/{site_id}/apps/{app_id}/variables/audit
POST   /api/v1/sites/{site_id}/variables/bulk-rotate
```

Request/Response Format

Variable Creation Request:

```
json
Copy code
{"key": "DATABASE_URL","value": "postgres://user:pass@host:5432/db","variable_type": "secret","description": "Primary
database connection string","overrides_site": true,"metadata": {"rotation_policy": "quarterly","tags": ["database",
"critical"],"owner_team": "backend"}}
```

Variable Response:

```
json
Copy code
{"id": "123e4567-e89b-12d3-a456-426614174000","key": "DATABASE_URL","value":
"postgres://user:pass@host:5432/db","variable_type": "secret","description": "Primary database connection
string","is_encrypted": true,"overrides_site": true,"created_at": "2024-01-01T00:00:00Z","updated_at": "2024-01-
01T00:00:00Z","created_by": "user-id","last_accessed_at": "2024-01-02T10:30:00Z","access_count": 42,"version":
3,"metadata": {"rotation_policy": "quarterly","tags": ["database", "critical"],"owner_team": "backend"}}
```

Resolved Variables Response:

```
json
Copy code
{"variables": {"DATABASE_URL": {"value": "postgres://app-specific-db","source": "app","variable_type":
"secret","overridden": true},"API_TIMEOUT": {"value": "30000","source": "site","variable_type": "config","inherited":
true},"REDIS_URL": {"value": "redis://app-cache","source": "app","variable_type": "secret","app_specific":
true}},"metadata": {"total_variables": 3,"inherited_count": 1,"overridden_count": 1,"app_specific_count":
1,"resolved_at": "2024-01-02T10:30:00Z"}}
```

API Implementation

```
python
Run CodeCopy code
# views/environment_views.pyfrom rest_framework import viewsets, status
from rest_framework.decorators import action
from rest_framework.response import Response
from django.utils import timezone
from ..models import SiteEnvironmentVariable, AppEnvironmentVariable
from ..services.environment_service import EnvironmentVariableService
from ..serializers import EnvironmentVariableSerializer

class SiteEnvironmentVariableViewSet(viewsets.ModelViewSet):
    serializer_class = EnvironmentVariableSerializer

    def get_queryset(self):
        return SiteEnvironmentVariable.objects.all()

    def perform_create(self, serializer):
        # The encryption happens automatically in the model's set_value method
        instance = serializer.save(created_by=self.request.user.id)

        # Log the creation
        self._log_access(instance, 'write', 'site')

    @action(detail=False, methods=['post'])def bulk_rotate(self, request, site_id=None):
        """Rotate multiple secrets at once"""# Implementation for bulk secret rotationreturn Response({'status':
```

```
'rotation_initiated'})


class AppEnvironmentVariableViewSet(viewsets.ModelViewSet):
    serializer_class = EnvironmentVariableSerializer

    def get_queryset(self):
        app_id = self.kwargs.get('app_id')
        return AppEnvironmentVariable.objects.filter(app_id=app_id)

    def perform_create(self, serializer):
        app_id = self.kwargs.get('app_id')
        instance = serializer.save(
            app_id=app_id,
            created_by=self.request.user.id
        )

        # Log the creation
        self._log_access(instance, 'write', 'app')

    @action(detail=False, methods=['get'])def resolved(self, request, site_id=None, app_id=None):
        """Get resolved variables with inheritance applied"""
        service = EnvironmentVariableService()
        resolved_vars = service.resolve_variables(
            app_id=app_id,
            user_id=request.user.id
        )

        return Response({
            'variables': resolved_vars,
            'metadata': {
                'total_variables': len(resolved_vars),
                'resolved_at': timezone.now().isoformat()
            }
        })

    @action(detail=True, methods=['post'])def rotate(self, request, site_id=None, app_id=None, pk=None):
        """Rotate a single secret"""
        variable = self.get_object()
        if variable.variable_type != 'secret':
            return Response(
                {'error': 'Only secrets can be rotated'},
                status=status.HTTP_400_BAD_REQUEST
            )

        # Implementation for secret rotation
        new_version = variable.version + 1# ... rotation logic ...return Response({
            'status': 'rotated',
            'new_version': new_version
        })
```

Performance Optimization

Caching Strategy

Multi-Level Caching:

- **L1 Cache (Application Memory)**: Frequently accessed variables cached in-memory

- **L2 Cache (Redis)**: Distributed cache for resolved variable sets
- **L3 Cache (Database)**: Optimized database queries with prepared statements

Simplified Cache Keys (No Environment):

```
css
Copy code
site_vars:{site_id}
app_vars:{site_id}:{app_id}
resolved_vars:{site_id}:{app_id}:{user_id}
user_permissions:{user_id}:{site_id}:{app_id}
```

Cache Invalidation:

- **Variable Updates**: Immediate invalidation of affected cache entries
- **Permission Changes**: Cache invalidation when user permissions change
- **Time-Based Expiry**: Short TTL for sensitive secrets (5 minutes), longer for config (1 hour)
- **Hierarchical Invalidation**: Site-level changes invalidate all dependent app caches

Key Management with django-encrypted-model-fields

```
python
Run CodeCopy code
# utils/encryption.pyimport os
from cryptography.fernet import Fernet
from django_tenants.utils import get_current_tenant

class TenantKeyManager:
    """Manage encryption keys per tenant"""    @staticmethoddef generate_tenant_keys(tenant_id: str) -> tuple[str, str]:
        """Generate primary and rotation keys for a tenant"""
        primary_key = Fernet.generate_key().decode()
        rotation_key = Fernet.generate_key().decode()

        return primary_key, rotation_key

    @staticmethoddef get_tenant_keys(tenant_id: str = None) -> list[str]:
        """Get encryption keys for current or specified tenant"""if not tenant_id:
            tenant = get_current_tenant()
            tenant_id = tenant.id if tenant else 'default'

        keys = []

        # Primary key
        primary_key = os.environ.get(f'TENANT_{tenant_id}_FERNET_KEY_1')
        if primary_key:
            keys.append(primary_key)

        # Rotation key (for seamless key rotation)
        rotation_key = os.environ.get(f'TENANT_{tenant_id}_FERNET_KEY_2')
        if rotation_key:
            keys.append(rotation_key)

        if not keys:
            # Fallback to default key for development
            default_key = os.environ.get('DEFAULT_FERNET_KEY')
```

```
            if default_key:
                keys.append(default_key)

        return keys

# Dynamic key loading based on current tenantdef get_encrypted_field_keys():
    """Dynamically get keys based on current tenant"""return TenantKeyManager.get_tenant_keys()

# Override the package's key loadingimport encrypted_model_fields.fields
encrypted_model_fields.fields.get_keys = get_encrypted_field_keys
```

Consequences

Positive Outcomes

Simplified Architecture:

- **No Environment Complexity**: Removing environment concept simplifies data model and APIs
- **Site-as-Environment**: Clear mapping where each site represents a complete environment
- **Cleaner Inheritance**: Simple two-tier inheritance (site → app) without environment complications
- **Reduced Complexity**: Fewer dimensions to consider in caching, permissions, and queries

Operational Benefits:

- **Environment Isolation**: Complete isolation between environments via separate sites
- **Simplified Deployment**: Deploy different environments as separate sites
- **Independent Scaling**: Each environment (site) can scale independently
- **Clear Boundaries**: No cross-environment variable leakage by design

Developer Experience:

- **Intuitive Model**: Sites as environments is more intuitive than nested environments
- **Simpler APIs**: Cleaner API endpoints without environment parameters
- **Reduced Confusion**: Less chance of misconfiguration with environment variables
- **Clear Separation**: Development, staging, production are clearly separate sites

Negative Consequences

Deployment Complexity:

- **Multiple Sites**: Need to create separate sites for dev/staging/prod environments
- **Site Management**: More sites to manage and maintain
- **Data Synchronization**: No built-in mechanism to sync configs between environment sites
- **Migration Overhead**: Moving from one environment to another requires site-level operations

Flexibility Limitations:

- **No Environment Variants**: Cannot easily have dev/staging variants within same site
- **Promotion Complexity**: Promoting configurations from dev to prod requires cross-site operations
- **Testing Challenges**: Testing with different environment configs requires separate sites

Risk Mitigation

Deployment Risks:

- **Site Provisioning**: Automated site creation for new environments
- **Configuration Templates**: Template-based configuration setup for new environment sites
- **Promotion Tools**: Build tooling for promoting configs between environment sites
- **Backup Strategies**: Comprehensive backup of all environment sites

Management Complexity:

- **Site Lifecycle Management**: Automated lifecycle management for environment sites
- **Configuration Drift**: Monitoring and alerting for configuration differences between sites
- **Access Control**: Consistent access control policies across environment sites

Alternatives Considered

Alternative 1: Keep Environment as Field

Approach: Maintain environment field as originally proposed.
Pros: More granular control, easier config promotion, single site for multiple environments **Cons**: Added complexity, potential for cross-environment data leakage, complex caching
Rejection Reason: User specifically requested to remove environment concept in favor of site-as-environment model.

Alternative 2: Environment as Separate Entity

Approach: Create Environment model linked to sites with variables linked to environments.
Pros: Maximum flexibility, clear environment management **Cons**: Additional complexity, three-tier hierarchy, goes against user's simplification request
Rejection Reason: Adds complexity that user wants to avoid.
This simplified architecture aligns with the user's vision of sites representing complete environments while maintaining all the security and inheritance features required for the platform.

Analytics Platform

Context

Taruvi Cloud needs an analytics platform that enables AI coding agents to programmatically create queries, dashboards, and visualizations through MCP servers. The platform must integrate seamlessly with the existing multi-tenant architecture while providing flexible frontend options.

Decision

We will implement a **Dual-Frontend Analytics Platform** with MCP-first API design, supporting both traditional visualization libraries and AI-generated frontend components.

Architecture Overview

Core Components

```
less
Copy code
AI Agents (MCP) → Analytics API → Query Engine → Data Sources
                        ↓
Frontend Options: [Grafana-style Library] OR [AI-Generated Components]
```

Data Architecture

Analytics Schema (Per Site)

```
sql
Copy code
-- Site schema: site_xyz_analytics-- Datasources (can reference site or app level data)
analytics_datasources:
  - id, name, description
  - datasource_type (site_wide, app_specific)
  - connection_config (which tables/schemas to query)
  - schema_metadata (cached field definitions)

-- Questions/Queries
analytics_questions:
  - id, name, description, datasource_id
  - query_type (sql, graphql, json_query)
  - query_definition (actual query string/object)
  - parameters_schema (JSON schema for filters)
  - result_schema (expected output structure)
  - cache_ttl, created_by_agent

-- Dashboards
analytics_dashboards:
  - id, name, description, app_id (nullable for site-wide)
  - layout_config (grid positions, sizes)
  - frontend_type (library, ai_generated)
```

```
  - created_by_agent

-- Dashboard Questions (many-to-many)
dashboard_questions:
  - dashboard_id, question_id, position, config
```

Query Engine Architecture

Abstraction Layer Design

```
class DataSourceConnector:
    """Unified interface for both JSONB and flat tables"""def execute_query(self, query: Query) -> QueryResult:
        if query.target_storage == 'jsonb':
            return self._execute_jsonb_query(query)
        elif query.target_storage == 'flat':
            return self._execute_sql_query(query)
        else:
            return self._execute_mixed_query(query)

    def get_schema_metadata(self, datasource: str) -> SchemaMetadata:
        """Return unified schema regardless of storage type"""pass
```

Query Types Supported

1. **Direct SQL**: For flat tables and complex joins
2. **JSONB Queries**: Optimized for entity-specific JSONB tables
3. **GraphQL-style**: Nested queries across relationships
4. **Mixed Queries**: Cross-storage-type queries using UNIONs

MCP Server Integration

Analytics MCP Endpoints

```
# Datasource Management
mcp_client.create_datasource({
    "name": "user_analytics",
    "type": "app_specific",
    "app_id": "app-123",
    "tables": ["app_123_users_jsonb", "app_123_orders_jsonb"],
    "schema_hint": "auto_detect"  # or explicit schema
})

# Question Creation
mcp_client.create_question({
    "name": "monthly_active_users",
    "datasource_id": "ds-456",
    "query": {
        "type": "jsonb_query",
        "target_table": "app_123_users_jsonb",
        "select": ["COUNT(DISTINCT id)"],
        "filters": [
            {"path": "data->last_login", "operator": ">", "value": "{{date_param}}"}
        ],
        "group_by": ["DATE_TRUNC('month', (data->>'created_at')::timestamp)"]
    },
    "parameters": [
        {"name": "date_param", "type": "date", "required": true}
```

```
    ]
})

# Dashboard Creation
mcp_client.create_dashboard({
    "name": "User Engagement Dashboard",
    "app_id": "app-123",
    "frontend_type": "ai_generated",  # or "library""questions": [
        {"question_id": "q-789", "position": {"x": 0, "y": 0, "w": 6, "h": 4}},
        {"question_id": "q-790", "position": {"x": 6, "y": 0, "w": 6, "h": 4}}
    ]
})
```

Frontend Architecture Options

Option A: Grafana-Style Library Frontend

```
typescript
Run CodeCopy code
// Pre-built React componentsimport { Dashboard, Chart, Table, Filter } from '@taruvi/analytics-ui'function
AnalyticsDashboard({ dashboardId }: { dashboardId: string }) {
  return (
    <Dashboard id={dashboardId}><Chart questionId="q-789" type="line" /><Table questionId="q-790" /><Filter
parameter="date_range" /></Dashboard>
  )
}
```

Pros:

- Fast development and consistent UX
- Battle-tested visualization components
- Easy maintenance and updates
- Built-in responsive design
- Standardized interaction patterns

Cons:

- Limited customization for unique use cases
- Generic look and feel
- Requires pre-building all chart types
- Less flexibility for AI agents to create novel visualizations

Option B: AI-Generated Frontend Components

```
typescript
Run CodeCopy code
// AI generates component code based on question schemaconst generatedComponent = await mcp_client.generate_frontend({
  question_id: "q-789",
  style_preferences: "modern, dark theme, interactive",
  component_type: "react",
  custom_requirements: "add export to PDF button"
})

// Returns actual React component code that can be dynamically loaded
```

Pros:

- Unlimited customization potential
- AI can create novel visualization types
- Tailored UX for specific use cases
- Can adapt to changing requirements automatically
- More engaging and unique dashboards

Cons:

- Higher complexity and potential bugs
- Requires robust AI code generation
- Harder to maintain consistency
- Security concerns with dynamic code execution
- Performance unpredictability

Hybrid Approach (Recommended)

Support both options with intelligent fallbacks:

```
python
Run CodeCopy code
class FrontendStrategy:
    def render_dashboard(self, dashboard_config):
        if dashboard_config.frontend_type == ”library”:
            return self.render_with_library(dashboard_config)
        elif dashboard_config.frontend_type == ”ai_generated”:
            try:
                return self.render_ai_generated(dashboard_config)
            except Exception:
                # Fallback to library if AI generation failsreturn self.render_with_library(dashboard_config)
        else:
            # Smart choice based on complexityreturn self.choose_optimal_frontend(dashboard_config)
```

API Design

RESTful Endpoints

```
bash
Copy code
# Datasources
GET/POST /api/v1/sites/{site_id}/analytics/datasources
GET/PUT/DELETE /api/v1/sites/{site_id}/analytics/datasources/{id}

# Questions
GET/POST /api/v1/sites/{site_id}/analytics/questions
GET/PUT/DELETE /api/v1/sites/{site_id}/analytics/questions/{id}
POST /api/v1/sites/{site_id}/analytics/questions/{id}/execute

# Dashboards
GET/POST /api/v1/sites/{site_id}/analytics/dashboards
GET/PUT/DELETE /api/v1/sites/{site_id}/analytics/dashboards/{id}

# App-specific endpoints
```

```
GET/POST /api/v1/sites/{site_id}/apps/{app_id}/analytics/questions
GET/POST /api/v1/sites/{site_id}/apps/{app_id}/analytics/dashboards
```

GraphQL Schema (Optional)

```
graphql
Copy code
type AnalyticsQuestion {id: ID!name: String!datasource: Datasource!queryDefinition: JSON!parameters: [Parameter!]!
  execute(filters: JSON): QueryResult!}type Dashboard {id: ID!name: String!questions: [DashboardQuestion!]!frontendType:
FrontendType!generatedComponent: String # AI-generated code}
```

Middle Layer Decision

Abstraction Layer (Recommended)

Create a middle layer that:

Pros:

- Unified query interface across storage types
- Query optimization and caching
- Security and access control enforcement
- Schema evolution handling
- Performance monitoring
- Query result transformation

Cons:

- Additional complexity
- Potential performance overhead
- Another layer to maintain
- Learning curve for developers

Direct Database Access Alternative

Pros:

- Maximum performance
- Direct SQL control
- Simpler architecture
- No translation overhead

Cons:

- AI agents need storage-type awareness
- Harder to maintain as schema evolves
- Security enforcement complexity
- No unified caching strategy

Decision: Implement Abstraction Layer for better AI agent experience and maintainability.

Performance Optimization

Caching Strategy

```
python
Run CodeCopy code
# Multi-level caching
L1_CACHE = "question_results:{question_id}:{filter_hash}"  # 5 min TTL
L2_CACHE = "datasource_schema:{datasource_id}"           # 1 hour TTL
L3_CACHE = "dashboard_layout:{dashboard_id}"             # 24 hour TTL
```

Query Optimization

- **Index Recommendations**: AI suggests optimal indexes based on query patterns
- **Query Plan Analysis**: Automatic EXPLAIN analysis with optimization suggestions
- **Result Set Streaming**: Large results streamed back for better UX
- **Parallel Execution**: Cross-entity queries executed in parallel

Security Model

Analytics-Specific Permissions

```
python
Run CodeCopy code
ANALYTICS_PERMISSIONS = [
    'analytics.datasource.create',
    'analytics.question.create',
    'analytics.question.execute',
    'analytics.dashboard.create',
    'analytics.results.export'
]

ANALYTICS_ROLES = [
    'Analytics Admin',      # Full access to analytics features'Dashboard Creator',    # Can create dashboards and
questions  'Report Viewer',       # Can view and execute existing questions'Data Explorer',       # Can create ad-hoc
queries
]
```

Data Access Control

- **Inherit Site/App Permissions**: Analytics queries respect existing data access rules
- **Query-Level Security**: Additional filters applied based on user permissions
- **Result Filtering**: Post-query filtering for sensitive data
- **Audit Logging**: All analytics access logged for compliance

Functions and Workflows


Context

Taruvi Cloud requires a comprehensive serverless functions platform that enables AI coding agents to programmatically create, deploy, and execute functions through MCP servers. The platform must support both custom code execution and external service integrations while maintaining the existing multi-tenant architecture and security model.

Requirements

Core Functionality

- **MCP Integration**: AI agents can create/manage functions via MCP servers
- **Multi-Function Types**: Custom code, system functions, templates, and proxy functions
- **External Integrations**: Seamless routing to n8n, Zapier, Make, and custom webhooks
- **Environment Variables**: Inherit site and app-level variables with function-specific overrides
- **Multiple Triggers**: HTTP, scheduled, database events, and queue-based triggers

System Requirements

- **Multi-Tenant Isolation**: Functions isolated per site with app-level scoping
- **Resource Management**: CPU/memory limits, execution timeouts, and concurrency controls
- **Security**: Isolated execution with controlled access and audit logging
- **Performance**: Sub-second invocation with warm worker pools
- **Reliability**: Retry logic, error handling, and health monitoring

Integration Requirements

- **Existing Architecture**: Leverage Django/Celery/ECS infrastructure
- **Environment Variables**: Access to existing site/app variable inheritance system
- **Access Control**: Integration with RBAC/ABAC security model
- **Monitoring**: Comprehensive logging and performance metrics

Current Constraints

- Django-based application with site-based schema isolation
- ECS Fargate compute with Celery background processing
- PostgreSQL database with existing multi-tenancy
- Redis caching and SQS message queuing

- Existing environment variable management system
- Security and compliance requirements (audit logging, encryption)

Decision

We will implement a **Hybrid Serverless Functions Platform** using Django/Celery architecture with support for both code execution and proxy-based external integrations, fully integrated with MCP servers and the existing environment variable system.

Architecture Overview

High-Level Components

```
scss
Copy code
AI Agents (MCP) → Functions Management API → Execution Router
                          ↓                          ↓
            Function Registry              [Code Executor]
                          ↓                          ↓
            Environment Resolution         [Proxy Executor]
                          ↓                          ↓
            Celery Task Queue → ECS Workers → External Services
                          ↓
            Results Cache & Audit Logs
```

Function Types Architecture

1. Custom Functions: User-written Python code executed in isolated environments **2. System Functions**: Pre-built library functions for common operations (Excel export, email, PDF generation) **3. Template Functions**: Scaffolded functions based on common patterns **4. Proxy Functions**: No-code integrations that route data to external services

Core Database Schema

```
python
Run CodeCopy code
# Core function modelclass ServerlessFunction(models.Model):
    app_id = models.UUIDField(db_index=True)  # Always app-scoped
    name = models.CharField(max_length=255)
    function_type = models.CharField(choices=[
        ('custom', 'Custom Function'),
        ('system', 'System Function'),
        ('template', 'Template Function'),
        ('proxy', 'Proxy Function')
    ])

    # Code execution config
    runtime = models.CharField(default='python3.11')
    handler = models.CharField(default='main.handler')
    source_code = EncryptedTextField()  # Base64 encoded zip
    requirements = models.TextField(blank=True)

    # Resource limits
    timeout_seconds = models.PositiveIntegerField(default=30)
    memory_limit_mb = models.PositiveIntegerField(default=128)
```

```
    # Environment inheritance
    environment_vars = models.JSONField(default=dict)
    inherit_site_vars = models.BooleanField(default=True)
    inherit_app_vars = models.BooleanField(default=True)

    # Triggers and state
    triggers = models.JSONField(default=list)
    is_active = models.BooleanField(default=True)
    created_by_agent = models.BooleanField(default=False)

# System functions libraryclass SystemFunction(models.Model):
    id = models.CharField(primary_key=True)  # e.g., 'excel_mapper_v1'
    name = models.CharField(max_length=255)
    category = models.CharField(choices=[
        ('data_processing', 'Data Processing'),
        ('communication', 'Communication'),
        ('integration', 'Integration')
    ])
    source_code = EncryptedTextField()
    requirements = models.TextField()
    input_schema = models.JSONField()  # Validation schema
    documentation = models.TextField()

# Proxy configuration for external integrationsclass ProxyConfiguration(models.Model):
    function = models.OneToOneField(ServerlessFunction)
    integration_type = models.CharField(choices=[
        ('n8n', 'n8n Webhook'),
        ('zapier', 'Zapier Webhook'),
        ('make', 'Make Scenario'),
        ('webhook', 'Generic Webhook')
    ])
    target_url = models.URLField()
    http_method = models.CharField(default='POST')
    auth_type = models.CharField(choices=[
        ('none', 'No Auth'), ('api_key', 'API Key'),
        ('bearer_token', 'Bearer Token'), ('oauth2', 'OAuth 2.0')
    ])
    auth_config = models.JSONField(default=dict)
    request_transformer = models.JSONField(default=dict)
    response_transformer = models.JSONField(default=dict)
    timeout_seconds = models.PositiveIntegerField(default=30)
    max_retries = models.PositiveIntegerField(default=3)

# Execution trackingclass FunctionInvocation(models.Model):
    function = models.ForeignKey(ServerlessFunction)
    invocation_id = models.CharField(unique=True)
    input_data = models.JSONField()
    output_data = models.JSONField(null=True)
    status = models.CharField(choices=[
        ('queued', 'Queued'), ('running', 'Running'),
        ('success', 'Success'), ('failed', 'Failed'), ('timeout', 'Timeout')
    ])
    execution_time_ms = models.PositiveIntegerField(null=True)
    environment_vars_snapshot = models.JSONField()
```

Environment Variables Integration

Inheritance Hierarchy: Function vars → App vars → Site vars

```
python
Run CodeCopy code
# Environment resolution example
{
    # Site-level (inherited if inherit_site_vars=True)"DATABASE_URL": "postgres://site-db",
    "SITE_NAME": "My Site",

    # App-level (inherited if inherit_app_vars=True, overrides site)  "API_TIMEOUT": "30000",
    "DATABASE_URL": "postgres://app-specific-db",  # Overrides site# Function-level (highest priority)"BATCH_SIZE":
"100",
    "DATABASE_URL": "postgres://function-specific-db"  # Final override
}
```

System Environment Variables (automatically injected):

```
python
Run CodeCopy code
{
    "TARUVI_SITE_ID": "site-123",
    "TARUVI_APP_ID": "app-456",
    "TARUVI_FUNCTION_NAME": "process_orders",
    "TARUVI_RUNTIME": "python3.11"
}
```

System Functions Library

Pre-built Functions Available for Installation:

```
python
Run CodeCopy code
SYSTEM_FUNCTIONS_REGISTRY = {
    'excel_mapper_v1': {
        'name': 'Excel Data Mapper',
        'description': 'Convert JSON data to Excel with custom mapping',
        'category': 'data_processing',
        'input_schema': {
            'datasource': 'string',
            'mapping': 'object',
            'query_filters': 'object'
        },
        'example_usage': {
            'datasource': 'app_123_users_jsonb',
            'mapping': {
                'Name': "data->>'name'",
                'Email': "data->>'email'"
            }
        }
    },

    'email_sender_v1': {
        'name': 'Email Sender',
        'description': 'Send templated emails with dynamic data',
        'category': 'communication',
        'input_schema': {
            'to_emails': 'array',
            'template': 'string',
            'data': 'object'
```

```
        }
    },

    'pdf_generator_v1': {
        'name': 'PDF Generator',
        'description': 'Generate PDFs from HTML templates',
        'category': 'data_processing'
    }
}
```

Proxy Functions Templates

External Integration Templates:

```
python
Run CodeCopy code
PROXY_TEMPLATES = {
    'n8n_webhook': {
        'integration_type': 'n8n',
        'http_method': 'POST',
        'auth_type': 'none',
        'request_transformer': {
            'static_values': {
                'source': 'taruvi',
                'timestamp': '{{context.invocation_time}}'
            }
        }
    },

    'zapier_webhook': {
        'integration_type': 'zapier',
        'http_method': 'POST',
        'request_transformer': {
            'field_mapping': {
                'data': 'data',
                'metadata.site_id': 'context.site_id'
            }
        }
    },

    'slack_webhook': {
        'integration_type': 'webhook',
        'request_transformer': {
            'field_mapping': {
                'text': 'data.message',
                'channel': 'data.channel'
            },
            'static_values': {
                'username': 'Taruvi Bot'
            }
        }
    }
}
```

MCP Server Integration Points

Function Management via MCP:

```
python
Run CodeCopy code
# Create custom function
mcp_client.create_function({
    "app_id": "app-123",
    "name": "process_user_signup",
    "source_code": "base64_encoded_zip",
    "requirements": "requests==2.28.0",
    "environment_vars": {"WELCOME_EMAIL_TEMPLATE": "signup_welcome"},
    "triggers": [{"type": "database", "table": "users", "events": ["INSERT"]}]
})

# Install system function
mcp_client.install_system_function({
    "app_id": "app-123",
    "system_function_id": "excel_mapper_v1",
    "name": "monthly_user_export",
    "environment_vars": {"EXPORT_SCHEDULE": "monthly"}
})

# Create proxy function
mcp_client.create_proxy_function({
    "app_id": "app-123",
    "name": "slack_notifications",
    "proxy_config": {
        "integration_type": "webhook",
        "target_url": "https://hooks.slack.com/services/...",
        "auth_type": "none"
    }
})

# Create from template
mcp_client.create_proxy_from_template({
    "app_id": "app-123",
    "template": "n8n_webhook",
    "target_url": "https://my-n8n.com/webhook/user-events"
})
```

Execution Architecture

Function Routing Logic:

1. **Function Type Detection**: Route to appropriate executor based on function type
2. **Environment Resolution**: Resolve inherited environment variables
3. **Execution Context**: Prepare isolated execution environment
4. **Resource Monitoring**: Apply CPU/memory/timeout limits
5. **Result Processing**: Handle success/failure and update metrics

Celery Task Integration:

```
python
Run CodeCopy code
@celery_app.taskdef execute_serverless_function(site_id, app_id, function_id, invocation_id):
    with tenant_context(site_id):
        function = ServerlessFunction.objects.get(id=function_id)
```

```
        if function.function_type == 'proxy':
            result = proxy_executor.execute(function, event, context)
        else:
            result = code_executor.execute(function, event, context)

        # Update invocation record with results
        update_invocation_record(invocation_id, result)
```

Execution Environments:

- **Code Functions**: Isolated Python subprocess with virtual environment
- **Proxy Functions**: HTTP client with authentication and retry logic
- **Resource Limits**: Memory, CPU, timeout, and file system constraints
- **Security**: Process isolation, network restrictions, and audit logging

Trigger System

Supported Trigger Types:

1. **HTTP Triggers**: REST endpoints for direct function invocation
2. bash
3. Copy code
4. `POST /api/v1/sites/{site_id}/apps/{app_id}/functions/{name}/invoke`
5. **Database Triggers**: Automatic execution on data changes
6. python
7. Run CodeCopy code
8. `{"type": "database", "table": "users_jsonb", "events": ["INSERT", "UPDATE"]}`
9. **Scheduled Triggers**: Cron-like scheduling
10. python
11. Run CodeCopy code
12. `{"type": "schedule", "cron": "0 9 * * 1", "timezone": "UTC"}`
13. **Queue Triggers**: SQS message-based invocation
14. python
15. Run CodeCopy code
16. `{"type": "queue", "queue_name": "user_processing"}`

Integration with Existing Systems

Multi-Tenant Architecture

- **Site-Level Isolation**: Functions deployed within site schemas
- **App-Level Scoping**: All functions belong to specific apps

- **Cross-Tenant Security**: No data leakage between sites
- **Schema Inheritance**: Access to site's entity tables and configurations

Security Integration

- **RBAC/ABAC**: Functions inherit app-level permissions
- **Environment Variables**: Secure access to encrypted secrets
- **Audit Logging**: All function executions logged for compliance
- **Process Isolation**: Sandboxed execution environments

Performance Integration

- **ECS Auto-scaling**: Function workers scale with demand
- **Redis Caching**: Function results and metadata cached
- **Connection Pooling**: Efficient database connection management
- **CDN Integration**: Static assets and results distributed globally

Consequences

Positive Outcomes

Developer Experience

- **Unified Platform**: Single platform for code and integrations
- **MCP Integration**: Seamless AI agent interaction
- **Rich Function Library**: Accelerated development with pre-built functions
- **Flexible Triggers**: Multiple ways to invoke functions
- **Environment Inheritance**: Simplified configuration management

Operational Benefits

- **Existing Infrastructure**: Leverages current Django/Celery/ECS stack
- **Cost Predictable**: No per-invocation charges unlike AWS Lambda
- **Security Consistent**: Same security model as main platform
- **Monitoring Integrated**: Unified observability across all services
- **Multi-Tenant Ready**: Built-in isolation and resource management

Business Benefits

- **Faster Time-to-Market**: Pre-built integrations and system functions
- **AI Agent Enablement**: Full programmatic control via MCP
- **Integration Ecosystem**: Easy connections to popular tools
- **Scalable Architecture**: Grows with customer needs
- **Vendor Independence**: No lock-in to specific cloud providers

Negative Consequences

Complexity Overhead

- **Function Type Management**: Multiple execution paths to maintain
- **Environment Resolution**: Complex inheritance logic
- **Security Surface**: More attack vectors to secure
- **Monitoring Complexity**: Multiple execution types to monitor
- **Testing Challenges**: Various function types require different test strategies

Operational Challenges

- **Resource Management**: Manual scaling vs. automatic serverless scaling
- **Cold Start Handling**: Function initialization time management
- **Error Debugging**: Distributed execution troubleshooting complexity
- **Version Management**: System function updates across tenants
- **Performance Optimization**: Balancing isolation with efficiency

Technical Debt Risks

- **Abstraction Leakage**: Proxy vs. code execution differences
- **Integration Maintenance**: External service API changes
- **Security Patching**: Regular updates to execution environments
- **Performance Bottlenecks**: Celery queue saturation under load
- **Storage Growth**: Function code and execution logs accumulation

Risk Mitigation Strategies

Performance Risks

- **Warm Worker Pools**: Pre-warmed Celery workers for faster startup
- **Function Caching**: Cache frequently used function code and environments
- **Queue Management**: Multiple priority queues for different function types
- **Resource Monitoring**: Proactive scaling based on queue depth and response times

Security Risks

- **Code Validation**: Static analysis and sandbox testing of function code
- **Environment Isolation**: Strict process and filesystem isolation
- **Secret Management**: Integration with existing encrypted variable system
- **Audit Compliance**: Comprehensive logging of all function activities

Operational Risks

- **Health Monitoring**: Function and integration endpoint health checks

- **Circuit Breakers**: Automatic disabling of failing integrations
- **Rollback Capability**: Version control and rollback for system functions
- **Documentation**: Comprehensive guides for function development and debugging

Alternatives Considered

Alternative 1: AWS Lambda Integration

Approach: Use AWS Lambda for serverless execution with API Gateway triggers.
Pros:

- True serverless scaling with automatic resource management
- No infrastructure management overhead
- Built-in monitoring and logging via CloudWatch

Cons:

- Vendor lock-in to AWS ecosystem
- Per-invocation pricing model increases costs
- Cold start latency issues (100-1000ms)
- Complex multi-tenant isolation requires separate Lambda functions
- Limited integration with existing Django-based architecture

Rejection Reason: Cost unpredictability, vendor lock-in, and poor integration with existing multi-tenant architecture.

Alternative 2: Docker-based Function Runtime

Approach: Deploy each function as separate Docker container on ECS.
Pros:

- Complete isolation between functions
- Language agnostic execution environment
- Easy resource allocation per function

Cons:

- Container startup overhead for each invocation
- Complex orchestration and networking
- Higher resource usage and costs
- Difficulty managing thousands of small containers

Rejection Reason: Resource inefficiency and operational complexity for the expected number of functions.

Alternative 3: WebAssembly (WASM) Runtime

Approach: Execute functions in WebAssembly runtime for isolation and performance.
Pros:

- Near-native performance with strong isolation

- Fast startup times (sub-millisecond)
- Language agnostic compilation target

Cons:

- Limited ecosystem and tooling maturity
- Restricted access to system resources
- Complex integration with existing Python ecosystem
- Limited debugging and monitoring capabilities

Rejection Reason: Technology immaturity and limited Python ecosystem support.

Alternative 4: Kubernetes Jobs for Functions

Approach: Execute functions as Kubernetes Jobs with resource limits.
Pros:

- Native resource isolation and management
- Built-in retry and failure handling
- Scalable job execution

Cons:

- Job startup overhead (seconds)
- Complex cluster management requirements
- Over-engineering for simple function execution
- Additional infrastructure complexity

Rejection Reason: Unnecessary complexity for current scale and requirements.

Implementation Strategy

Phase 1: Core Function Platform (Month 1-2)

- **Basic Infrastructure**: Django models, Celery tasks, execution engine
- **Custom Functions**: Python code execution with resource limits
- **HTTP Triggers**: REST API endpoints for function invocation
- **Environment Integration**: Site/app variable inheritance
- **MCP Basic Integration**: Create and invoke functions via MCP

Phase 2: System Functions Library (Month 3)

- **Pre-built Functions**: Excel export, email sender, PDF generator
- **Installation System**: MCP endpoints for system function installation
- **Function Registry**: Searchable catalog of available system functions
- **Documentation**: Usage examples and API documentation

Phase 3: Proxy Functions (Month 4)

- **Proxy Architecture**: HTTP client with authentication and retry logic
- **Integration Templates**: n8n, Zapier, Make, Slack, Discord templates
- **Configuration Management**: Template-based proxy setup
- **Health Monitoring**: Integration endpoint health checks

Phase 4: Advanced Features (Month 5-6)

- **Advanced Triggers**: Database events, scheduled jobs, queue triggers
- **Performance Optimization**: Function caching, warm pools, monitoring
- **Security Hardening**: Enhanced isolation, audit logging, compliance
- **Operational Tools**: Function debugging, performance analytics, alerting

Success Metrics

- **Adoption Rate**: 70% of new apps use at least one serverless function
- **MCP Integration**: 80% of functions created via MCP servers
- **Performance**: 95% of function invocations complete within 2 seconds
- **Reliability**: 99.9% function execution success rate
- **Integration Usage**: 50% of functions use proxy integrations

This architecture provides a comprehensive serverless functions platform that leverages existing infrastructure while offering powerful capabilities for AI agents and developers to create, deploy, and manage both code-based and integration-based functions.

Logical Architecture